TFG: Métodos Monte Carlo basados en cadenas de Markov

José Jiménez

JIMENEZLUNA.COM

Primera edición, Enero 2015

Abstract: Markov Chain Monte Carlo (or shortly MCMC) is a powerful method for sampling from high dimensional probability distributions. Here we present a swift introduction to the theory behind these methods as well as several applications of the Bayesian MCMC framework. These include, among others Bayesian Mixture Models, Bayesian Image Analysis or Text Mining. Implementations of solutions regarding these problems can be found in several programming languages.

Índice general

1. Motivación del problema

Los métodos MCMC (Markov Chain Monte Carlo) surgen de la necesidad de simular el comportamiento de variables aleatorias y de estimar parámetros de las funciones de densidad/probabilidad de las mismas. El gran impulso a estas técnicas se las da mayormente (pero no únicamente) el enfoque estadístico bayesiano, donde la inferencia se realiza sobre lo que se denomina una función a posteriori, que denominaremos $\pi(\theta|x)$.

Las siglas MCMC vienen marcadas por las cadenas de Markov y por la integración Monte Carlo. La inferencia bayesiana, en multitud de ocasiones necesita integrar sobre distribuciones de dimensión muy elevada (en muchas ocasiones, con cientos de parámetros). Existen métodos numéricos aproximados que producen buenas soluciones, pero que no escalan bien con la dimensión, siendo en muchos casos, computacionalmente intratables. De manera poco formal, la aplicación de técnicas MCMC consta de dos pasos:

1. Generar una muestra $X_1, ..., X_n$ mediante una cadena de Markov cuya distribución estacionaria sea la buscada.
2. Tomar medias muestrales (integración Monte Carlo) y realizar inferencias sobre la muestra anteriormente citada

Algunos términos no quedan explicados a estas alturas. No obstante, esta estructura general que quedará aclarada en los siguientes capítulos permite resolver muchos problemas.

1.1 Problemática en inferencia bayesiana

Como hemos mencionado, en inferencia bayesiana, se realiza inferencia sobre una distribución a posteriori $\pi(\theta|x)$. Bajo un enfoque bayesiano, no existe diferencia conceptual entre parámetros y valores observables, es decir, son en su totalidad cantidades aleatorias. Denominemos por x a los datos observados, y θ al conjunto de parámetros (nótese que pueden ser ambos multidimensionales). En estadística bayesiana, la función a posteriori puede expresarse de la siguiente manera, aplicando el teorema de Bayes:

$$\pi(\theta|x) = \frac{L(x|\theta)\pi(\theta)}{\int L(x|\theta)\pi(\theta)} \tag{1.1}$$

Donde $L(x|\theta)$ no es más que la función de verosimilitud de los datos asumiendo que siguen

una distribución parametrizada y $\pi(\theta)$ es lo que se denomina una distribución a priori. Ésta última expresa en inferencia bayesiana toda información previa o creencia que se tiene sobre el comportamiento de los parámetros del modelo. O lo que es lo mismo, en inferencia bayesiana los parámetros son también variables aleatorias, cuya información se incluye en un análisis a posteriori, tomando también como referencia la muestra x ensayada.

En muchísimas ocasiones (por no decir casi todas), esta distribución a posteriori se conoce únicamente salvo constante multiplicativa (que fuerza a que la distribución integre 1 sobre su dominio Ω). Por esta razón, generalmente se suele notar de la siguiente manera:

$$\pi(\theta|x) \propto L(x|\theta)\pi(\theta) \tag{1.2}$$

Es decir, se dice que la función a posteriori es proporcional al numerador de 1.1. Una vez que tenemos determinada esta distribución, el enfoque bayesiano realiza inferencia sobre esperanzas de funciones de la misma.

1.2 Cálculo de esperanzas

Como acabamos de mencionar, la problemática ahora surge de estimar $E[f(x)]$ sobre la distribución a posteriori. En la mayoría de ocasiones trabajaremos sobre espacios paramétricos elevados, donde las soluciones analíticas pasan a ser un dolor de cabeza, y donde las numéricas no escalan bien. (De hecho, se pueden realizar simulaciones donde se comprueba que en espacios paramétricos con más de 20 dimensiones estos métodos suelen fallar estrepitósamente.)

En el resto del trabajo seguimos una hoja de ruta encaminada a aprender cómo resolver estos problemas mencionados de acuerdo con los pasos citados en el punto anterior. Antes de ello, realizaremos un muy breve repaso a algunos métodos de simulación de variables aleatorias básicos (que serán útiles a lo largo del trabajo), luego recordaremos algunas propiedades básicas de las cadenas de Markov que nos darán soporte teórico sobre lo que queremos hacer, y para finalizar los conceptos previos hablaremos muy brevemente de integración Monte Carlo.

Una vez finalizada la introducción, explicamos los métodos MCMC en general, dando varios algoritmos generales para resolución de dichos problemas (con ejemplos sencillos), algunos fundamentos teóricos para sustentar validación de resultados y algunos métodos de simulación más avanzados.

La última parte (y la más pesada) del trabajo corresponde a varias aplicaciones prácticas de estos métodos. Algunos ejemplos son tratamiento de imágenes o mixturas. En esta parte del trabajo especialmente se hará un uso intensivo de lenguajes de programación, como pueden ser R o Python 2.7. Las librerías utilizadas en cada parte se detallan para posterior reproducibilidad. Todo el código del trabajo queda disponible en el repositorio `github.com/hawk31/MCMC_tfg` bajo licencia MIT.

2. Conceptos previos

En este capítulo repasamos algunas ideas previas para la mejor comprensión de las aplicaciones posteriores. Estas ideas básicas servirán para esbozar el marco de actuación bayesiano ante un problema determinado.

2.1 Muestreo por rechazo

El muestreo por rechazo proporciona una manera muy eficiente de simular valores de una variable aleatoria $f(x)$. Suponemos en este sentido que no tenemos un algoritmo sencillo (como puede ser mediante inversión) para generar valores aleatorios de dicha distribución. La idea es utilizar una distribución instrumental $g(x)$, que acote absolutamente a $f(x)$, es decir $f(x) < Mg(x)$, con $M > 1$ y de la que sepamos generar valores aleatorios de una manera fácil.

El algoritmo funciona de la siguiente manera:

- Muestrear un valor x de $g(x)$ y u perteneciente a una U(0,1).
- Si $u < \dfrac{f(x)}{Mg(x)}$, aceptar x como valor aleatorio de $f(x)$. En caso contrario rechazar el valor y volver al primer paso.

Nótese que este método supone que tanto f como g son evaluables y que se conoce la constante multiplicativa M hasta cierto punto. Este método, aunque eficiente, tiene algunas desventajas, como que la distribución g debe ser parecida en curtosis y simetría a f, de manera que en el segundo paso del algoritmo se acepte una proporción de veces aceptable. En caso de que esto no ocurra, desechamos muchas muestras. En este sentido, existen métodos un poco más complejos (Muestreo por rechazo adaptativo) que tratan este problema de una manera más atractiva.

El muestreo por rechazo es precursor directo del algoritmo de Metrópolis-Hashtings, que además tiene la ventaja de escalar mucho mejor con la dimensionalidad de la simulación.

Podemos realizar una pequeña simulación de valores aleatorios de procedentes de una N(0,1) utilizando como distribución instrumental una Cauchy(0,2), con constante $M = 3$. (Nota: si se desea ver la adecuación de esta distribución instrumental bastaría con representarla)

Ejercicio 2.1
```
generarec<- function(n, M)
{
  resul<- numeric(n)
  inten<- integer(n)
  for (j in 1:n)
  {
    i=0
    repeat
    {
    i=i+1
    x=rcauchy(1,0,2)
    u=runif(1)
    fx=dnorm(x)
    if (u< fx/(M*dcauchy(x,0,2))){
     resul[j]<- x
     inten[j]<- i
     break
}
   }
  }
 return(list(x=resul,nint=inten))

}
```
■

2.2 Cociente de uniformes

Este método no requiere de una densidad instrumental, a diferencia del método de muestreo por rechazo ya explicado. Supongamos que queremos muestrear de una densidad f, y que tenemos una muestra bivariante uniforme U,V, que satisface la siguiente propiedad:

$$0 < u < \sqrt{f(v/u)} \tag{2.1}$$

Entonces V/U tiene densidad proporcional a f. Un pequeño ejemplo de aplicación, suponiendo que quisiéramos simular valores de una $N(0,1)$, usando U,V uniformes de parámetros adecuados.

Ejercicio 2.2
```
u = runif(10000,0,1)
v = runif(10000,-1,1)
alea = numeric(0)

for(i in 1:length(u)){
  if(u[i]<sqrt(dnorm(v[i]/u[i]))){alea = c(alea,v[i]/u[i])}
}
```
■

2.3 Integración Monte Carlo

El método Monte Carlo provee una estructura general para estimar integrales definidas finito-dimensionales . El marco general de actuación funciona de la siguiente manera:

Suponemos que queremos estimar $I = \int_\Omega f(x)dx$. Denotamos por $V = \int_\Omega 1dx$, es decir, al volumen del dominio sobre el que queremos integrar. El enfoque clásico o inocente Monte Carlo supone que tomamos muestras suficientemente grandes $X_1, ..., X_n$ de una distribución uniforme i.i.d. definida en Ω.

Por tanto, I podria ser aproximada por

$$\hat{I} \approx V \frac{1}{n} \sum_i f(X_i) \tag{2.2}$$

Esto puede realizarse porque la ley de los grandes números asegura que dicha estimación converge en media al valor de la integral verdadero. De hecho, podríamos incluso (ya que tenemos una muestra iid) estimar varianzas o cuasivarianzas muestrales para obtener errores de estimación:

$$Var[\hat{I}] \approx \frac{1}{n-1} \sum_i (f(X_i) - \frac{\hat{I}}{V})^2 \tag{2.3}$$

Esta estimación de la varianza permitiría obtener intervalos de confianza aproximados para una integral concreta, usando los puntos críticos adecuados.

Hemos explicado el método Monte Carlo básico para la estimación de una integral. Lo que nos interesará es usar dicho método para la estimación de esperanzas de una distribución concreta (en nuestro caso distribuciones a posteriori.) Por tanto, podemos particularizar este método:

$$E[f(x)] \approx \frac{1}{n} \sum_i X_i \tag{2.4}$$

Existen métodos de reducción de la varianza para los métodos Monte Carlo (y que por tanto permiten estimaciones más precisas), pero asumen que las muestras tomadas son *iid*. Esta es una situación bastante irreal, y por tanto no dedicaremos más tiempo a detallar estas particularidades.

Lo que realizaremos a continuación es un pequeño ejemplo en código `R` de cómo aplicar integración Monte Carlo. Supongamos que queremos estimar $\int_0^e 3e^{-3x}dx$. El valor exacto de dicha integral es $1 - e^{-3e} \approx 0{,}99713$

Ejercicio 2.3

```
#!usr/bin/env RScript
set.seed(99)
x = runif(10^5,0,exp(1))
v = exp(1)
f = function(x){3*exp(-3*x)}

alea = f(x)
(est = v*mean(alea))
[1] 0.9965623
```

■

2.4 Muestreo por importancia

Hemos supuesto en el apartado anterior, que para estimar una integral cualquiera (una esperanza), podríamos muestrear de la distribución objetivo f y tomar medias muestrales. No obstante, este procedimiento no se puede aplicar en todos los casos. El muestreo por importancia

utiliza la siguiente idea: Utilizaremos una distribución instrumental g con unas condiciones determinadas, generaremos valores aleatorios de g y reponderaremos la integral convenientemente.

Supongamos que queremos estimar $E_{f(x)}[h(x)]$. Lo que hacemos realmente es lo siguiente:

$$E_{f(x)}[h(x)] = \int_\Omega h(x)f(x)dx = \int_\Omega h(x)f(x)\frac{g(x)}{g(x)} = E_{g(x)}[h(x)f(x)/g(x)] \tag{2.5}$$

Se hace énfasis en este apartado en los subíndices para indicar con respecto a quién se toma esperanza. Por tanto, la solución es similar a la del apartado anterior: generamos suficientes valores aleatorios de la distribución g, aplicamos $\frac{h(x)f(x)}{g(x)}$ y tomamos la media muestral de esos valores.

De manera parecida a como pasaba en el muestreo por rechazo, la calidad de la estimación de esta integral dependerá en gran medida de la elección de la distribución instrumental. En términos generales, esta debe tener una forma similar a f, pero con colas más gruesas para asegurar que todo el dominio queda correctamente representado.

Como en apartados anteriores, podemos hacer una simulación rápida suponiendo que queremos estimar $E[cos(X)]$, donde $X \sim N(0,1)$, usando como distribución instrumental $Y \sim Cauchy(0,2)$

Ejercicio 2.4
```
#!usr/bin/env RScript
y = rcauchy(10^5,0,2)
f = cos(y)*dnorm(y)/dcauchy(y,0,2)
mean(f)

mean(cos(rnorm(10^5)))
```
■

Este realmente es un método eficiente para reducir la varianza en las estimaciones Monte Carlo, bajo las suposiciones que se han realizado sobre g. No obstante, en muchas ocasiones, el cálculo de tantas funciones puede ser computacionalmente pesado y por tanto poco eficiente.

2.5 Cadenas de Markov

De manera rápida, recordemos que una cadena de Markov es un proceso estocástico donde la probabilidad de un estado solo depende del estado inmediatamente anterior. De manera más formal, una cadena de Markov $X_1, ..., X_t$ es una secuencia de variables aleatorias (no i.i.d.) que cumple la siguiente propiedad:

$$P[X_{t+1}|X_t, X_{t-1}, X_{t-2}, ...] = P[X_{t+1}|X_t] \tag{2.6}$$

Generalmente, estas probabilidades no dependen de t en general, en cuyo caso se denominan homogéneas.

Nos preguntamos a continuación en qué medida afecta X_0 a X_t. Para responder a esta cuestión, necesitamos la distribución $P^t[X_t|X_0]$, en la que no interviene ningún valor intermedio. Las buenas noticias es que bajo condiciones de regularidad laxas, la cadena irá olvidando según vaya avanzando su estado inicial, convergiendo hacia lo que se denomina distribución estacionaria $\phi(x)$ (que no depende ni de t ni del estado inicial X_0). Esto será de vital importancia para los

algoritmos que veremos en el capítulo siguiente.

De esta manera, podemos ir entreviendo parte de la solución del problema: Si conseguimos construir una cadena de Markov tal que su distribución estacionaria sea la distribución que estamos interesados en muestrear, tendremos facilidad para hacer estimaciones realizando integración Monte Carlo.

Normalmente los valores generados por cadenas de Markov no son i.i.d., por lo que necesitaremos resultados parecidos a los garantizados por el teorema central del límite para realizar estimaciones, que veremos más adelante.

2.6 Propiedades de las Cadenas de Markov

Hemos afirmado en el apartado anterior que para que una cadena de Markov converja a una distribución estacionaria (o invariante) $\phi(x)$, debía cumplir varias condiciones. En particular, debe cumplir las siguientes propiedades:

- La cadena debe ser irreducible. Es decir, para todo punto inicial de la cadena, la cadena puede alcanzar un conjunto de estados cualquiera con probabilidad positiva en un número finito de iteraciones. Más formalmente, una cadena de Markov se denomina irreducible si para todo i, j, existe $t > 0$, tal que $P_{ij}(t) > 0$, donde $P_{ij}(t)$ es la matriz de transición de la cadena de un estado i a un estado j.
- La cadena debe ser aperiódica. Es decir, la cadena no puede estar oscilando en un número finito de iteraciones entre dos estados.
- La cadena debe ser positivo-recurrente. Esto puede ser expresado en términos de que si un valor de la cadena, por ejemplo X_0, es de la distribución estacionaria, la cadena convergerá a dicha distribución.

En los métodos MCMC nuestro objetivo ya es muestrear de una distribución estacionaria $\phi(x)$, por lo que si construímos X, nos bastaría demostrar que es irreducible para demostrar que es positivo-recurrente. Estas cadenas tienen además algunas propiedades muy interesantes. Conectando con lo explicado de integración Monte Carlo, la mayoría de estimaciones que realizamos sobre una cadena construída con estas condiciones se resume en una media ergódica:

$$\overline{f_n} = \frac{\sum_t X_t}{n} \tag{2.7}$$

Por tanto, sería de interés conocer el comportamiento asintótico de estas estimaciones. Resulta que para una cadena positivo-recurrente aperiódica, la distribución estacionaria que buscamos coincide con la distribución límite de la cadena, y por tanto, las estimaciones marcadas convergen a sus respectivas esperanzas. Esto queda expresado de manera más formal:

Theorem 2.6.1 (Teorema ergódico.) Con una cadena con condiciones anteriores, si $E[f(X)] < \infty$, entonces $P[\overline{f_n} \to E[f(X)]] = 1$. Es decir, que converge en media cuadrática a su esperanza.

Este teorema es de vital importancia en la aplicación de los métodos MCMC que explicaremos en el siguiente capítulo. No obstante, no proporciona información sobre cuán larga debe ser la cadena para poder realizar estimaciones precisas. Generalmente en estas cadenas se suele descartar un número alto de valores al principio de la misma, en lo que se denomina periodo de calentamiento o burn-in. Este problema será tratado más adelante, junto con algunos resultados de tasas de convergencia.

3. Markov Chain Monte Carlo

En este capítulo partimos de lo explicado anteriormente y proporcionamos métodos generales para los métodos MCMC. Habíamos explicado que en inferencia bayesiana, el objeto de estudio comprende una distribución a posteriori $\pi(\theta|x)$ (de la que queremos muestrear para hacer inferencia). En este sentido, se explicó que como no se tenía un método general para muestrear de distribuciones de altas dimensiones, se construye una cadena de Markov cuya distribución estacionaria sea nuestra distribución a posteriori, y usando integración Monte Carlo tomamos medias muestrales de funciones de la cadena para obtener información.

Por tanto, lo primero y más importante es aprender cómo construir cadenas con esta propiedad. Por ello, existen dos familias de algoritmos bastante usados, a los que haremos referencia extendida a lo largo del trabajo: el algoritmo de Metropolis-Hastings y el muestreo de Gibbs.

3.1 Algoritmo de Metropolis-Hastings

El algoritmo de Metrópolis-Hastings es un muestreo por rechazo generalizado, donde los valores aleatorios se toman de distribuciones escogidas razonablemente y corregidas de tal manera que se comporten asintóticamente como valores de la distribución objetivo.

El algoritmo de Metropolis-Hastings funciona de la siguiente manera: supongamos que tenemos una cadena de Markov en un estado X_t y queremos actualizar al valor X_{t+1}. Para ello utilizamos una distribución candidata $q(.|X_t)$, que genera un valor aleatorio propuesto que denotaremos y. Es importante notar que esta distribución puede depender del estado actual de la cadena.

Con estas condiciones, y se acepta con probabilidad $\alpha(X_t, Y)$, donde este último término es:

$$\alpha(X_t, Y) = min\left(1, \frac{\pi(y)q(x|y)}{\pi(x)q(y|x)}\right) \tag{3.1}$$

Si se acepta este valor, y pasa a ser el siguiente estado de la cadena X_{t+1}. En caso contrario, $X_{t+1} = X_t$.

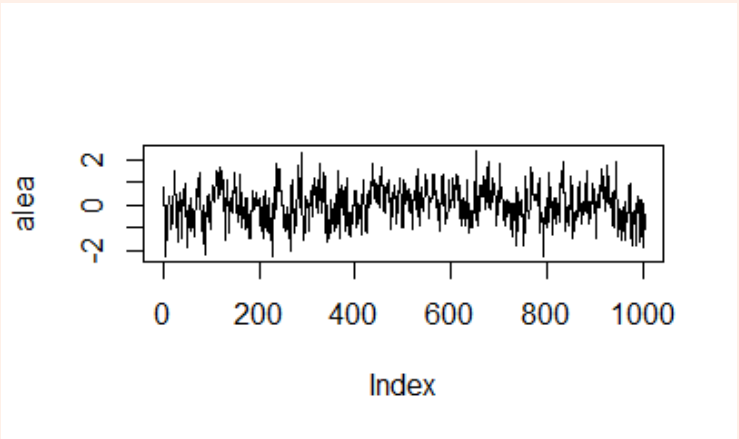
La distribución $q(.|.)$ puede tener cualquier forma y la cadena proporcionada por el algoritmo convergerá a la distribución estacionaria $\pi(x)$. Si bien esta condición es suficiente, la distribución proposición debe tener la misma dimensión que la estacionaria, y ser capaz de generar valores que se acepten. En otro caso la cadena puede pasar periodos largos de tiempo en un mismo estado. Nótese que en el momento que X_t ya pertenezca a la distribución estacionaria, todos los valores siguientes $X_{t+1}, X_{t+2}, ...$ pertenecerán igualmente a la misma distribución.

Ejercicio 3.1 Para ilustrar cómo funciona el algoritmo de Metropolis-Hastings, utilizaremos un ejemplo de juguete. Supongamos que queremos muestrear valores de una distribución $N(0,1)$,utilizando como distribución propuesta $N(X_t, 0{,}5)$. Una posible implementación podría ser la siguiente.

```
n = 1000
alea = numeric(n)
alea[1]=0   #mu
for (i in 2:n)
{
  y = rnorm(1,alea[i-1],0.5)
  u = runif(1)
  alpha = min(1,(dnorm(y)*dnorm(alea[i-1],y,0.5))/(dnorm(alea[i-1])*
                                                    dnorm(y,alea[i-1],0.5)))
  if(u<alpha) alea[i]=y
  else alea[i]=alea[i-1]
}
```

En la práctica, se suele realizar lo que se denomina 'traza' de la simulación, que no es más que un gráfico de línea para comprobar la convergencia y el mezclado de la cadena.

```
plot(alea,type="l")
```



■

3.1.1 Distribuciones proposición

Como hemos notado antes, cualquier distribución proposición $q(.|.)$ es suficiente para que la cadena converja a su distribución estacionaria. No obstante, la elección adecuada de esta distribución y su relación con la estacionaria garantizará una convergencia más rápida hacia esta

última. Es más, aún garantizada convergencia, la cadena podría 'mezclar' de manera lenta (es decir, que se mueva lentamente por el soporte de la distribución objetivo). Por ello, escoger una distrubución propuesta adecuada se convierte en un problema. Existen varias formas canónicas que detallamos a continuación.

3.2 Algoritmos de muestreo

Tradicionalmente, la literatura sobre MCMC ha hablado de muestreadores y algoritmos. No obstante, aunque aquí seguimos esas convenciones, conviene no olvidar que unos muestreadores no excluyen a otros. Como veremos un poco más adelante, es común combinarlos para construir una cadena de Markov con mejor rendimiento. Es por ello que tiene más sentido denominarlos como 'actualizadores MCMC' más que muestreadores. Estos conceptos quedarán más asentados cuando expliquemos el muestreador de Gibbs.

3.2.1 Algoritmo de Metropolis

El original propuesto por Metropolis en 1950 supone que utilizamos distribuciones proposición simétricas (esto es $q(X|Y) = q(Y|X)$). Con esta condicion 3.1, pasa a ser:

$$\alpha(X_t, Y) = min\left(1, \frac{\pi(y)}{\pi(x)}\right) \tag{3.2}$$

3.2.2 Paseo aleatorio Metropolis

Si $q(x,y) = f(y-x)$ para una densidad particular f, el algoritmo se denomina de paseo aleatorio Metropolis. Se suelen usar densidades f simétricas, para aceptar con probabilidades idénticas a las del apartado anterior.

3.2.3 Muestreo con independencia

Supongamos a continuación que $q(x,y) = f(y)$, y por tanto los posibles candidatos a la cadena se generan independientemente del estado X_t de la cadena. En este caso, la probabilidad de aceptación puede escribirse como:

$$\alpha(x,y) = min\left(1, \frac{w(y)}{w(x)}\right) \tag{3.3}$$

donde $w(x) = \pi(x)/f(x)$ es lo que se denomina el peso de importancia. Este método guarda grandes similitudes con las ideas mostradas en el muestreo por importancia. La diferencia esencial entre los dos métodos es que el muestreador por importancia genera la masa de probabilidad sobre puntos con pesos altos, escogiéndolos frecuentemente. En contraste, el muestreo de independencia construye masa de probabilidad en puntos con altos pesos, permaneciendo en ellos largos periodos de tiempo.

Ejercicio 3.2 Supongamos que queremos muestrear de una distribución Gamma de parámetros a, b cualesquiera usando el muestreador de independencia. Utilizaremos una distribución $N(a/b, a/b^2)$ como propuesta. Una posible implementación podría ser:

```
#! /usr/bin/env RScript
gammaSampler<-function (n, a, b)
{
  mu <- a/b
  sig <- sqrt(a/b^2)
```

```
  vec <- numeric(n)
  x <- mu
  vec[1] <- x
  for (i in 2:n) {
    can <- rnorm(1, mu, sig)
    aprob <- min(1, (dgamma(can, a, b)/dgamma(x,
                                              a, b))/(dnorm(can, mu, sig)/dnorm(x,
                                                                                mu, sig)
    u <- runif(1)
    if (u < aprob) vec[i] <- can
    else vec[i]<- vec[i-1]
  }
  vec
}
```

Nótese que hemos utilizado la media de la distribución objetivo como primer valor de la cadena para intentar conseguir una convergencia rápida. En la práctica esto no suele ser posible. ■

En la práctica, el muestreador de independencia puede funcionar muy bien o muy mal. Para que funcione bien, $q(.)$ debería ser una buena aproximación a la distribución objetivo, aunque generalmente basta con que tenga colas más pesadas. En efecto, si $q(.)$ no cumple esta condición, puede pasar mucho tiempo atascado en las colas de la distribución objetivo, proporcionando mal rendimiento.

3.2.4 Actualizando en bloques

En los ejemplos que hemos estado realizando hasta ahora, hemos utilizado distribuciones con un número bajo de parámetros, para facilitar la comprensión. En la práctica desafortunadamente nos encontramos con situaciones en las que podemos llegar a tener cientos de parámetros. Es por ello que se han desarrollado procedimientos para actualizar cada uno de los componentes 'en bloque' o secuencialmente. El método original propuesto por Metropolis se denomina single-component Metropolis-Hastings.

Supongamos que tenemos que actualizar $X_{,1},....,X_{,h}$. Es decir, tenemos que actualizar h elementos, secuencialmente. Denotemos por $X_{.-i}$ al mismo vector sin el elemento i-ésimo. Para cada i de la iteración $t+1$, actualizamos cada componente utilizando Metropolis-Hastings. El candidato $Y.i$ se genera mediante distribución propuesta $q_i(Y_{.i}|X_{t.i},X_{t.-i})$. Esto quiere decir que la distribución propuesta puede depender del resto de componentes del vector y de sí misma en la iteración anterior. (Teniendo en cuenta que si se han actualizado algunos antes en la misma iteración $t+1$ pueden actualizarse los siguientes teniendo esto en cuenta). Por tanto, para cada componente del vector tendremos una probabilidad de aceptación al estilo anterior, con modificaciones pertinentes:

$$\alpha(X_{.-i},X_{.i},Y_{.i}) = min\left(1,\frac{\pi(Y_{.i}|X_{.-i})q_i(X_{.-i}|Y_{.i},X_{.-i})}{\pi(X_{.i}|X_{.-i})q_i(Y_{.-i}|X_{.i},X_{.-i})}\right) \tag{3.4}$$

En esta expresión $\pi(.|X_{.-i})$ se denomina distribución totalmente condicionada. Esto no es más que la expresión fijando todos los parámetros menos el de interés, tratándolos como constantes. El uso de distribuciones totalmente condicionadas juega un papel muy importante en los métodos MCMC y en especial inferencia bayesiana, donde se aplican en modelos condicionales de

dependencia (ya los veremos en aplicaciones), y que en la mayoría de situaciones ofrecen una simplificación de 3.4.

3.3 Muestreo de Gibbs

El muestreador de Gibbs cobra especial importancia justo después de este último apartado. El mismo nos proporciona una manera general y sencilla de actualizar los componentes del vector de parámetros mencionado utilizando distribuciones totalmente condicionadas. Utilizar Metropolis-Hastings para cada componente tiene la desventaja evidente de que tenemos que buscar h densidades propuesta, una para cada uno de los componentes. El muestreador de Gibbs propone lo siguiente: bajo las mismas condiciones que antes, la distribución propuesta de cada uno de los componentes será:

$$q_i(Y_{.i}|X_{.i}, X_{.-i}) = \pi(Y_{.i}|X_{.-i}) \tag{3.5}$$

Es decir, se propone utilizar la distribución totalmente condicionada de un parámetro al resto de parámetros (valga la redundancia) como distribución propuesta. Esto, como hemos dicho antes pasa por fijar el resto de parámetros y tratarlos como constantes (valores iniciales o los de la iteración anterior). Finalmente, actualizamos la densidad resultante (univariante, esta vez) con el método que más convenga, ya sea Metropolis-Hastings u otro.

Este método tiene gran aplicabilidad en todos los ejemplos prácticos que veremos al final del trabajo, ya que proporciona una manera directa de actualizar los componentes.

Ejercicio 3.3 Para ejemplificar cómo funciona el muestreador de Gibbs, supongamos que queremos generar valores aleatorios de la siguiente distribución con 3 variables.

$$f(y_1, y_2, y_3) \propto exp\left(-(y_1 + y_2 + y_3 + \theta_{12}y_1y_2 + \theta_{13}y_1y_3 + \theta_{23}y_2y_3\right) \tag{3.6}$$

Para aplicar el muestreador de Gibbs es necesario determinar las distribuciones totalmente condicionadas de y_1, y_2, y_3. De esta manera:

$$f(y_1|y_2, y_3) \propto exp(-y_1(1 + \theta_{12}y_2 + \theta_{13}y_3)) \tag{3.7}$$

Nótese que podemos ignorar aquellos términos en los que no aparece y_1 porque son constantes multiplicativas. Con un poco de ojo, y teniendo en cuenta que aquí y_2 e y_3 son constantes nos damos cuenta que $y_1|y_2, y_3 \sim Exp(1 + \theta_{12}y_2 + \theta_{13}y_3)$. Por simetría y de manera idéntica, $y_2|y_1, y_3 \sim Exp(1 + \theta_{12}y_1 + \theta_{23}y_3)$ y $y_3|y_1, y_2 \sim Exp(1 + \theta_{12}y_1 + \theta_{23}y_2)$. El ejemplo no estaría completo sin una simulación pertinente, para la cual supondremos unos θ_{ij} cualesquiera.

```
niter = 10^4

theta = c(2,3,1/2)
sim = matrix(NA,nrow=niter,ncol=3)
sim[1,] = c(0.4,1.2,0.3) ## Valores iniciales (cualesquiera)

for(i in 2:niter){
  sim[i,1] = rexp(1,1+theta[1]*sim[i-1,2]+theta[2]*sim[i-1,3])
  sim[i,2] = rexp(1,1+theta[1]*sim[i,1]+theta[3]*sim[i-1,3])
  ## Nótese que aquí ya hemos actualizado el primer
  ## componente y lo usamos para acelerar convergencia.
  sim[i,3] = rexp(1,1+theta[2]*sim[i,1]+theta[3]*sim[i,2]) ## Idem
```

```
}
```

En este codigo hacemos uso de `rexp`, la función ya implementada en R para generar valores según una exponencial determinada. No obstante, no tenemos por qué tener un algoritmo eficiente en alguna de estas distribuciones condicionadas, por lo que podriamos usar Metropolis-Hastings o alguna de las técnicas ya descritas. ■

3.4 Muestreo por rechazo adaptativo (ARS)

Aprovecharemos que ya hemos explicado el muestreador de Gibbs para explicar una técnica bastante usada en conjunto. Hasta ahora, cuando pretendíamos usar el muestreador de Gibbs suponíamos que teníamos que simular distribuciones totalmente condicionadas al resto, y que para ello podriamos recurrir a Metropolis-Hastings, al muestreo por rechazo o a cualquier otra técnica que proporcione buenos resultados.

Si bien las anteriores técnicas funcionan bien, es necesario determinar en todas y cada una de ellas una densidad propuesta de la que generar los valores que luego serán aceptados o rechazados con una determinada probabilidad. El método que a continuación se propone es una técnica del tipo 'caja negra' en el sentido de que no necesita esta determinación realizando una serie de hipótesis sobre la densidad de la que se pretende muestrear.

Empezamos definiendo algunos conceptos. Supongamos que queremos muestrear de una densidad $g(x)$, la cual podemos conocer hasta una constante multiplicativa (constante de integración), continua y diferenciable en todo su dominio. Supongamos además que $h(x) = ln\ g(x)$ es cóncava en todo D. Sea $T_k = (x_1, ..., x_k)$, k puntos de abcisa en los que han sido evaluados tanto $h(x)$ como $h'(x)$. A continuación definimos un contorno de rechazo en T_k como $exp\ u_k(x)$, donde $u_k(x)$ es una función lineal definida a trozos definida por las tangentes de $h(x)$ en T_k. Para $j = 1, ..., k-1$, las tangentes de x_j y x_{j+1} intersecan en:

$$z_j = \frac{h(x_{j+1}) - h(x_j) - x_{j+1}h'(x_{j+1}) + x_j h'(x_j)}{h'(x_j) - h'(x_{j+1})} \tag{3.8}$$

Esta función es una aproximación via contorno lineal por encima de $g(x)$.Por tanto, para $x \in [z_{j-1}, z_j]$, definimos:

$$u_k(x) = h(x_j) + (x - x_j)h'(x_j) \tag{3.9}$$

donde z_0 es el extremo inferior de D (o $-\infty$ si no está acotado inferiormente) y z_k es el extremo superior de D (o ∞ si no está acotado superiormente.). Con estos conceptos, definimos la densidad:

$$s_k(x) = \frac{exp\ u_k(x)}{\int_D exp\ u_k(x')dx'} \tag{3.10}$$

Finalmente, definimos la función restricción inferior en T_k como $expl_k(x)$, donde $l_k(x)$ es una función lineal definida a trozos por debajo de $g(x)$, formada entre abcisas adyacentes en T_k. Para $x \in [x_j, x_{j+1}]$:

$$l_k(x) = \frac{(x_{j+1} - x)h(x_j) + (x - x_j)h(x_{j+1})}{x_{j+1} - x_j} \tag{3.11}$$

La concavidad de $h(x)$ asegura que $l_k(X) \leq h(x) \leq u_k(x)$ en todo el dominio. Tras todas estas definiciones, podemos poner en marcha el algoritmo.

1. Inicializar la abcisa en T_k. Si D no está acotado por la izquierda, escogemos x_1 tal que $h'(x_1) > 0$. Si D está no acotado por la derecha, entonces escogemos x_k tal que $h'(x_k) < 0$.
2. Calculamos las funciones $u_k(x), s_k(x)$ y $l_k(x)$ para estos k puntos.
3. Usamos $s_k(x)$ para muestrear un valor x^* y w de una uniforme unidad.
4. Si $w \leq exp\,(l_k(x^*) - u_k(x^*))$ aceptar x^*.
5. En otro caso, si $w \leq exp\,(h(x^*) - u_k(x^*))$, aceptar x^*. En cualquier otro caso rechazarlo.
6. Si $h(x^*)$ ha tenido que ser evaluado en este último test, incluímos x^* en T_k para formar T_{k+1} y volvemos al segundo paso hasta que tengamos n puntos muestreados.

Este algoritmo es el que se implementa en la mayoría de software especializado dedicado al muestreador de Gibbs, como pueden ser `BUGS` (o sus múltiples variantes) o `JAGS`, por lo que tiene gran importancia. En capítulos posteriores haremos uso de estas herramientas para algunas tareas. En `R` existe una implementación en el paquete `ars`.

Ejercicio 3.4 Supongamos que queremos muestrear valores de una Beta(2,3) mediante Adaptive Rejection Sampling. Utilizamos el paquete mencionado.

```
library(ars)
#Muestrear de una Beta(2,3)
n=20
f2<-function(x,a,b){(a-1)*log(x)+(b-1)*log(1-x)}
f2prima<-function(x,a,b){(a-1)/x-(b-1)/(1-x)}
mysample2<-ars(20,f2,f2prima,x=c(0.3,0.6),m=2,
               lb=TRUE,xlb=0,ub=TRUE,xub=1,a=2,b=3)
mysample2
```

■

3.5 Otras consideraciones

En este apartado consideraremos algunos aspectos a tener en cuenta cuando se están realizando simulaciones MCMC en general, desde cómo determinar unos valores iniciales adecuados hasta reglas para determinar un número de iteraciones adecuado.

3.5.1 Valores iniciales

Si bien como hemos dicho los valores iniciales X_0 no influyen en la distribución estacionaria de la cadena, si éstos no se escogen adecuadamente algunas cadenas pueden mezclar lentamente, teniendo que descartar muchas iteraciones como calentamiento. Una manera de determinar un buen valor inicial es correr varias cadenas simultáneamente con distintos valores iniciales (si se tiene la suficiente potencia computacional en paralelo). Si se dispone de información previa sobre la distribución estacionaria, un buen punto de partida podría ser la mediana.

3.5.2 Calentamiento

El número de iteraciones de calentamiento, es decir, todas aquellas previas que se descartan por no pertenecer a la distribución de la que se pretende muestrear depende tanto de los valores iniciales como de la tasa de convergencia de la cadena (hablamos de esto último en el siguiente apartado). En mayor medida depende de cuán cerca estén las distribución propuesta y la estacionaria. La manera en la que se suele determinar el número m de iteraciones que se

debe descartar se suele realizar mediante análisis gráfico de la traza de la cadena. Existen en la literatura algunos estimadores de m analíticos, pero no suelen verse con demasiada soltura en las aplicaciones probablemente porque el análisis gráfico proporciona la suficiente información.

Otra cuestión relacionada es cuándo parar la cadena para realizar estimaciones Monte Carlo. La medida más usual para ello es correr como se ha sugerido antes varias cadenas en paralelo, de tal manera que si convergen rápidamente todas hacia una región determinada podemos asumir que ya ha alcanzado su distribución estacionaria y correr esta vez una cadena suficientemente larga para realizar estimaciones.

Generalmente, las iteraciones pertenecientes al calentamiento se descartan en las estimaciones de medias y varianzas pues pueden sesgarlas.

3.5.3 Análisis de la salida

Como se ha venido explicando mediante pinceladas hasta ahora, una salida de una simulación Monte Carlo proporciona suficiente información para:

- Realizar un gráfico de líneas de la traza para comprobar convergencia y calentamiento. Descartar estas últimas iteraciones en los siguientes análisis.
- Estimar medias y varianzas de la cadena.
- Podríamos incluso estimar intervalos de confianza, usando las estimaciones de varianza ahora mismo tomadas o de manera más sencilla, tomando los percentiles $\frac{\alpha}{2}$ y $1-\frac{\alpha}{2}$ de la simulación.
- Rizando un poco el rizo, podriamos estimar las distribuciones marginales mediante alguna función núcleo $K(X_{.i})$. Una elección bastante corriente para esta función es la distribución totalmente condicionada de $X_{.i}|X_{.-i}$, como se explicó en el muestreador de Gibbs.

3.5.4 Tasa de convergencia

Se dice que una cadena de Markov X es geométricamente ergódica si es aperiódica positiva recurrente y existe un $\lambda \in (0,1)$ y una función $V(.)$ tal que se cumple:

$$\sum_j |P_{ij} - \pi(j)| \leq V(i)\lambda \tag{3.12}$$

El λ más pequeño para el que exista una función V que satisfaga la condición anterior se denomina tasa de convergencia. Lo notaremos por λ^*. Para entender mejor las implicaciones de las cadenas geométricamente ergódicas recurrimos al análisis espectral. Esto se escapa ampliamente del objetivo del trabajo, pero diremos que para las cadenas geométricamente ergódigas el principal autovalor es $\lambda_0 = 1$ y el resto (finitos) están acotados por el círculo unidad. La velocidad a la que converge la cadena o λ^* es por tanto dependiente del segundo autovalor más grande.

3.5.5 Estimación de la varianza

Una de las consecuencias más importantes de las cadenas ergódicas es que permite la existencia de resultados del tipo teorema central del límite para medias ergódicas de la forma:

$$\overline{f_n} - E\left[f(X)\right] \rightarrow N(0, \sigma^2) \tag{3.13}$$

Dicha convergencia es en distribución. Por tanto es de vital importancia estimar correctamente σ^2. Si bien pueden utilizarse estimadores simples como los propuestos en integración Monte Carlo, se han propuesto algunas alternativas más sofisticadas.

Batch means

La idea detrás de este procedimiento es correr una cadena de Markov en $N = mn$ iteraciones, con n suficientemente grande. Notemos por:

$$Y_k = \frac{1}{n} \sum_{i=(k-1)n+1}^{kn} f(X_i) \tag{3.14}$$

con $k = 1, \ldots, m$. De esta manera, una buena estimación de σ^2 podría venir dada por:

$$\hat{\sigma^2} \approx \frac{n}{m-1} \sum_{k=1}^{m} \left(Y_k - \overline{f_N}\right) \tag{3.15}$$

Ejercicio 3.5 Una posible implementación general de batch means podría ser la siguiente:

```
n=1000
m=5

cadena = rexp(n*m,rate = 1/3)

batch.means <- function(chain,f,m){
  fN = mean(f(chain))
  n = length(chain)/m
  li = numeric(m)
  for(i in 1:m){
    li[i]=mean(f(chain[((i-1)*n+1):(i*n)]))
  }
  sigma = n/(m-1)*sum((li-fN)^2)
  return(sigma)
}

batch.means(cadena,identity,5)
```

■

Estimadores de ventana

Otra opción para estimar σ^2 es usar la función de autocovarianzas muestral. Sin embargo, este método produce estimadores inconsistentes según aumenta el retardo i, por lo que se propone una versión truncada del estimador, que puede expresarse de la siguiente manera:

$$\sigma^2 \approx \hat{\gamma}_0 + 2\sum_{i}^{\infty} w_n(i)\hat{\gamma}_i \tag{3.16}$$

donde los $w_n(i)$ son pesos verificando $|w_n(i)| < 1$ y $\sum_i w_n(i) = 1$. Una posible implementación rápida podria ser la siguiente:

Ejercicio 3.6

```
cadena = rexp(5000,rate = 1/3)

window.est <- function(cadena,weights){
  aut = as.numeric(acf(cadena,plot=F,type="covariance")$acf)
  sigma = aut[1]+2*sum(pesos*aut[2:(length(pesos)+1)])
  return(sigma)
```

```
}

pesos = rep(1/12,12)
window.est(cadena,pesos)
```

4. Modelos graficos y DAGs

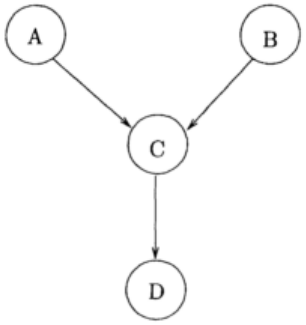
4.1 Digrafos acíclicos

Este capítulo será el más corto dedicado a aplicaciones. Los modelos gráficos se usan cada vez más frecuencia en modelos bayesianos en los que se aplica MCMC. Las relaciones entre las variables del modelo pueden representadas haciendo que los nodos en un grafo representen esas variables, y los ejes entre los nodos anteriores representando la relación (o ausencia) en términos de independencia condicional.

Estos grafos, generalmente se presentan con una estructura jerárquica, con aquellos nodos que ejercen una influencia más directa sobre los datos colocados en la parte inferior, y según va disminuyendo dicha influencia colocando superiormente. Estos grafos proporcionan una manera fácil de interpretar la estructura condicional del modelo, simplificando la implementación del algoritmo MCMC en cuestión, indicando qué variable ejerce influencia sobre qué otra en su distribución.

Estos grafos se denotan DAGs (Directed Acyclic Graphs). Consisten en una colección de nodos y ejes dirigidos, donde la dirección de estos ejes determina el sentido de la dependencia entre variables. Los nodos, a su vez, pueden ser de dos tipos: círculos si se trata de una variable desconocida y que por tanto hay que estimar simulando, o cuadrados si se trata de una variable conocida. Los DAGs se caracterizan por ser acíclicos, es decir, no podemos volver al mismo nodo de partida sea cual sea. Un ejemplo de DAG podría ser el de la figura 4.1.

Figura 4.1: Un DAG simple



En este ejemplo, los nodos A y B se denominan padres de C. De igual manera, D es hijo de C. De esta manera, queda representado que la variable C depende de A y B. De igual manera el valor de D depende de C, pero además es condicionalmente independiente de A y B, dado un valor de C. Esta independencia condicional queda representada mediante la ausencia de ejes entre A y B hacia D. De esta manera, esta afirmación podría notarse como:

$$D \perp A, B|C \tag{4.1}$$

En general, para cualquier nodo v, podríamos escribir:

$$v \perp \text{antepasados de } v|\text{padres de } v \tag{4.2}$$

Este tipo de construcción permite extraer la distribución conjunta de todos los parámetros del modelo de la siguiente manera (suponiendo $v \in V$):

$$P(V) = \prod_{v \in V} P(v|\text{padres de } v) = P(A)P(B)P(D|C)P(C|A,B) \tag{4.3}$$

4.2 Grafo de independencia condicional

Si bien esto es cierto, los DAG no proporcionan información exhaustiva sobre todas las relaciones posibles del modelo. Supongamos que queremos estimar A conociendo C: en ese caso el valor de B sería útil en esa tarea, y por tanto A no es condicionalmente independiente de B dado C. Por tanto, B figura en la distribución condicional de A. Toda esta información puede ser recogida mediante lo que se denomia un grafo de independencia condicional. Éste se obtiene 'moralizando' el DAG. Esto implica 'casar' a todos los padres añadiendo un eje entre los padres de cada nodo v del grafo. Por último, abandonamos la direccionalidad de todos los ejes. Este grafo sí proporciona información exhaustiva sobre las relaciones entre los parámetros del modelo. Por ejemplo, dado un nodo v, la distribución completa condicional del mismo, condicionado al valor del resto se expresa como:

$$P(v|\text{resto}) \propto P(v|\text{padres de } v) \prod_{u \in C_v} P(u|\text{padres de } u) \tag{4.4}$$

donde C_v denota el conjunto de hijos del nodo v. Con esto tenemos información suficiente para aplicar los métodos MCMC que ya conocemos, siendo usado con mayor frecuencia el muestreador de Gibbs dada este tipo de situaciones.

4.3 Ejemplos de aplicación

Si bien todas las aplicaciones posteriores que se van a ver utilizan este tipo de modelos gráficos de una manera u otra, he considerado oportuno introducir algún que otro modelo real, cuyo trasfondo teórico no tenemos por qué conocer (Para eso están los siguientes capítulos). En muchos de los casos, la distribución de los nodos nos será dada, en otras la función de densidad puede estar incompleta hasta una constante de integración, o puede ser difícil muestrear de ellas (para lo que ya se ha visto una buena técnica puede ser ARS). No obstante, existe software especifico para este tipo de modelos. Entre ellos podemos citar a la familia `BUGS` o a `JAGS`. Usaremos `OpenBUGS` en esta sección para ilustrar cómo se trabaja desde un punto de vista bayesiano usando MCMC.

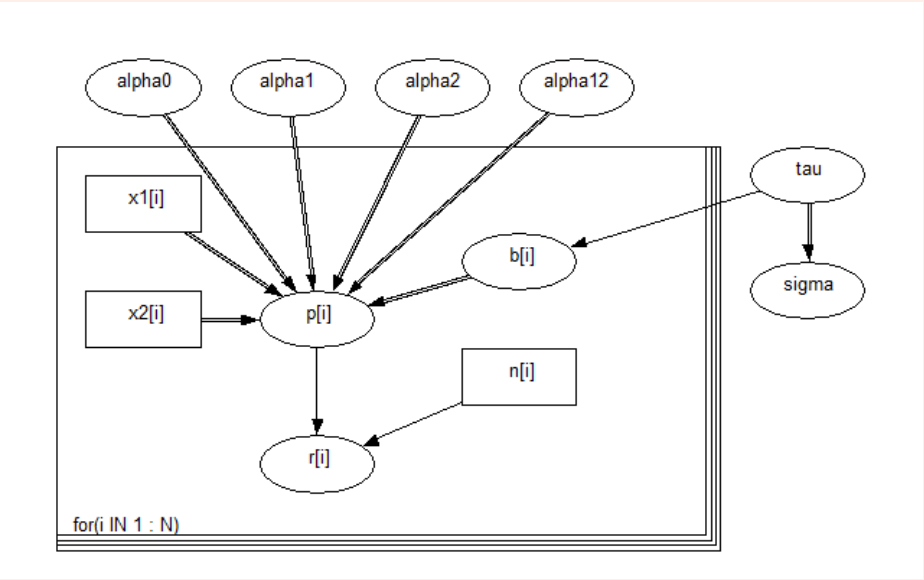
Ejercicio 4.1 Tomaremos uno de los ejemplos guiados de este software para ilustrar cómo se suele trabajar con estos modelos. Los datos pertenecen a un estudio realizado por Crawler (1973). Supongamos que tenemos $n = 21$ platinas en las que se siembra una semilla, de tal manera que realizamos un estudio factorial 2×2 dependiendo del tipo de semilla x_1 y tipo de extracto de raíz x_2. Notemos por n_i por el número total de semillas en la platina i, y r_i al total de semillas germinadas en esa misma platina. Los datos son los siguientes:

r_i	n_i	x_1	x_2
10	39	0	0
23	62	0	0
23	81	0	0
26	51	0	0
17	39	0	0
5	6	0	1
53	74	0	1
55	72	0	1
32	51	0	1
46	79	0	1
10	13	0	1
8	16	1	0
10	30	1	0
8	28	1	0
23	45	1	0
0	4	1	0
3	12	1	1
22	41	1	1
15	30	1	1
32	51	1	1
3	7	1	1

Tratamos lo siguiente como un modelo de regresión logística de efectos aleatorios. Para aplicar las técnicas de simulación conocidas, necesitamos conocer la distribución totalmente condicionada de los parámetros que figuran en el siguiente modelo:

$$r_i \sim Binomial(n_i, p_i) \tag{4.5}$$
$$logit(p_i) = \alpha_0 + \alpha_1 x_{1i} + \alpha_2 x_{2i} + \alpha_{12} x_{1i} x_{2i} + \varepsilon_i \tag{4.6}$$
$$\varepsilon_i \sim N(0, \tau) \tag{4.7}$$

Se incluye un término de interacción en el modelo por completitud. La función logit puede ser cualquier función logística. Sobre la distribución de $\alpha_0, \alpha_1, \alpha_2, \alpha_1 2$ y τ diremos que les asignamos distribuciones independientes no informativas. Lo realmente interesante para enlazar con el apartado anterior es enseñar cuál es el grafo de independencia completa, que proporciona suficiente información como para implementar el modelo en BUGS.



La siguiente sintaxis debería resultar natural viendo el gráfico anterior. La distribución independiente de los parámetros anteriormente citados viene especificada.

```
model
{
for( i in 1 : N ) {
r[i] ~ dbin(p[i],n[i])
b[i] ~ dnorm(0.0,tau)
logit(p[i]) <- alpha0 + alpha1 * x1[i] + alpha2 * x2[i] +
alpha12 * x1[i] * x2[i] + b[i]
}
alpha0 ~ dnorm(0.0,1.0E-6)
alpha1 ~ dnorm(0.0,1.0E-6)
alpha2 ~ dnorm(0.0,1.0E-6)
alpha12 ~ dnorm(0.0,1.0E-6)
tau ~ dgamma(0.001,0.001)
sigma <- 1 / sqrt(tau)
}
```

Después de 6000 iteraciones (5000 después de las 1000 de calentamiento), podemos

obtener estimadores de los parámetros de los modelos y lo que es más interesante, de los p_i, lo que permitiría realizar inferencia sobre las platinas.

```
mean sd MC_error val2.5pc median val97.5pc start sample
alpha0 -0.5477 0.1827 0.004813 -0.9062 -0.5465 -0.1859 1001 5000
alpha1 0.07567 0.3104 0.00861 -0.5517 0.08125 0.678 1001 5000
alpha12 -0.8459 0.4191 0.0129 -1.697 -0.8407 -0.05176 6001 5000
alpha2 1.341 0.2692 0.007286 0.8032 1.342 1.873 11001 5000
p[1] 0.3255 0.05479 9.903E-4 0.2157 0.3283 0.427 1001 15000
p[2] 0.3681 0.04677 3.957E-4 0.2786 0.3666 0.4646 1001 15000
p[3] 0.3232 0.04496 8.851E-4 0.2346 0.3239 0.4085 1001 15000
p[4] 0.4297 0.0608 0.001492 0.3234 0.4246 0.5584 1001 15000
p[5] 0.3935 0.05681 8.655E-4 0.2928 0.3894 0.5169 1001 15000
p[6] 0.6996 0.06647 8.907E-4 0.5648 0.6992 0.8325 1001 15000
p[7] 0.7009 0.04182 5.327E-4 0.6194 0.7008 0.783 1001 15000
p[8] 0.7229 0.04434 9.283E-4 0.6385 0.7221 0.8098 1001 15000
p[9] 0.6615 0.04951 5.689E-4 0.5556 0.6652 0.7495 1001 15000
p[10] 0.633 0.04801 0.00104 0.5334 0.636 0.7191 1001 15000
p[11] 0.6996 0.06186 7.993E-4 0.5756 0.6985 0.825 1001 15000
p[12] 0.415 0.07302 7.875E-4 0.2801 0.4116 0.5728 1001 15000
p[13] 0.3714 0.06263 8.701E-4 0.2498 0.3727 0.4927 1001 15000
p[14] 0.3564 0.06586 0.001123 0.2251 0.3578 0.4814 1001 15000
p[15] 0.44 0.06221 9.817E-4 0.3251 0.4378 0.5673 1001 15000
p[16] 0.3609 0.08497 0.001398 0.1831 0.3642 0.5211 1001 15000
p[17] 0.4659 0.08556 0.001889 0.2766 0.4749 0.6133 1001 15000
p[18] 0.5262 0.0581 5.036E-4 0.409 0.5267 0.6375 1001 15000
p[19] 0.5122 0.06323 6.911E-4 0.3842 0.5132 0.6341 1001 15000
p[20] 0.5673 0.05791 9.225E-4 0.4578 0.5666 0.6838 1001 15000
p[21] 0.5057 0.08076 0.001095 0.3359 0.509 0.6593 1001 15000
```

Otros métodos de optimización para los parámetros producen resultados similares a los ofrecidos por MCMC (máxima verosimilitud, mínimos cuadrados). Más información en el tutorial `Seeds` de `OpenBUGS`.

■

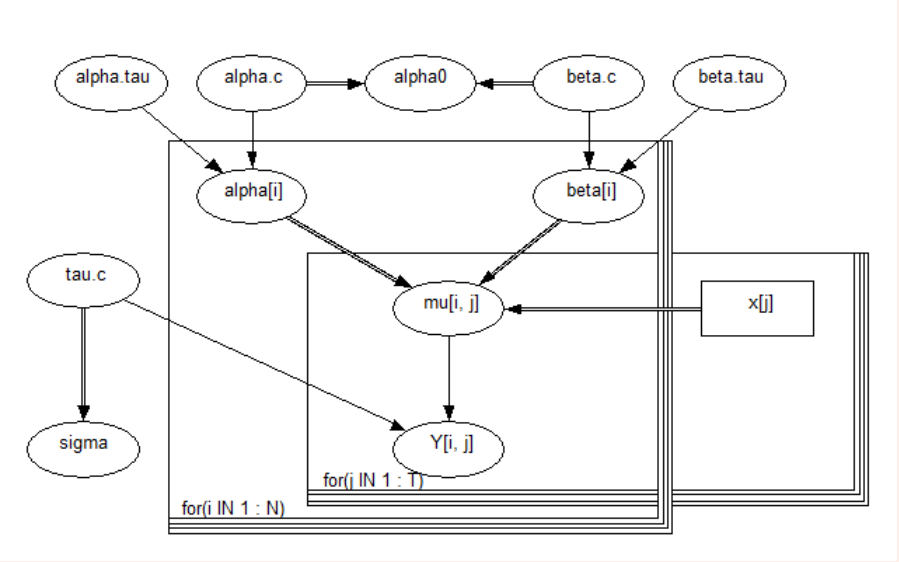
Ejercicio 4.2 En este ejemplo, también de la documentación de `OpenBUGS`, usaremos un modelo normal jerárquico para volver a ilustrar el procedimiento de simulación para estimación de los parámetros.

Supongamos que estamos realizando un experimento sobre $n = 30$ ratas a las que se les mide el peso semanalmente durante un periodo de 5 semanas. Los datos se pueden consultar en la documentación del software o en el repositorio del trabajo.

Proponemos un modelo de crecimiento lineal con efectos aleatorios. Eso supone que la especificación condicional de nuestro modelo es la siguiente:

$$Y_{ij} \sim N(\alpha_i + \beta_i(X_j - X_{bar}), \tau_c) \tag{4.8}$$
$$\alpha_i \sim N(\alpha_c, \tau_\alpha) \tag{4.9}$$
$$\beta_i \sim N(\beta_c, \tau_\beta) \tag{4.10}$$

Donde $X_{bar} = 22$, y todos los τ representan la precisión (1/varianza) de una distribución normal. De manera similar al ejemplo anterior, a $\alpha_c, \tau_\alpha, \beta_c, \tau_\beta$ se les dan distribuciones a priori independientes. Para visualizar cómo implementar este modelo, recurrimos al grafo de independencia completa:



Pasamos a la implementación en `BUGS`.

```
model
{
for( i in 1 : N ) {
for( j in 1 : T ) {
Y[i , j] ~ dnorm(mu[i , j],tau.c)
mu[i , j] <- alpha[i] + beta[i] * (x[j] - xbar)
}
alpha[i] ~ dnorm(alpha.c,alpha.tau)
beta[i] ~ dnorm(beta.c,beta.tau)
}
tau.c ~ dgamma(0.001,0.001)
```

```
sigma <- 1 / sqrt(tau.c)
alpha.c ~ dnorm(0.0,1.0E-6)
alpha.tau ~ dgamma(0.001,0.001)
beta.c ~ dnorm(0.0,1.0E-6)
beta.tau ~ dgamma(0.001,0.001)
alpha0 <- alpha.c - xbar * beta.c
}
```

Corremos el modelo para muestrear de los parámetros un número razonable de veces, descartando calentamiento (10000 en nuestro caso). Después de hacerlo, algunos de los resultados sobre los parámetros son los siguientes:

```
mean sd MC_error val2.5pc median val97.5pc start sample
mu[1,1] 154.9 4.324 0.04283 146.5 154.9 163.3 1 10000
mu[1,2] 197.4 3.166 0.03059 191.2 197.4 203.6 1 10000
mu[1,3] 239.9 2.677 0.02329 234.6 239.9 245.1 1 10000
mu[1,4] 282.4 3.181 0.02558 276.1 282.3 288.6 1 10000
mu[1,5] 324.8 4.346 0.03566 316.2 324.8 333.3 1 10000
mu[2,1] 149.0 4.503 0.05022 140.3 149.0 158.0 1 10000
mu[2,2] 198.4 3.259 0.03566 192.1 198.4 204.8 1 10000
mu[2,3] 247.8 2.708 0.02792 242.4 247.8 253.1 1 10000
alpha[1] 239.9 2.677 0.02329 234.6 239.9 245.1 1 10000
alpha[2] 247.8 2.708 0.02792 242.4 247.8 253.1 1 10000
alpha[3] 252.4 2.705 0.02555 247.1 252.4 257.7 1 10000
alpha[4] 232.6 2.675 0.02376 227.4 232.6 237.8 1 10000
alpha[5] 231.6 2.704 0.02725 226.3 231.7 237.0 1 10000
```

■

Con este ejemplo concluímos el capítulo. Los siguientes van dedicados a aplicaciones de la teoría explicada a campos particulares, como el análisis de imagen o la imputación múltiple. No obstante, el fundamento teórico para resolver los problemas sí será explicado, a diferencia de los modelos de este capítulo.

5. Mixturas gaussianas

En este capítulo veremos una de las aplicaciones más interesantes que tienen las técnicas MCMC. Dentro del campo del aprendizaje no supervisado, podemos considerar las mixturas como parte de los algoritmos de clustering o conglomerados, donde dado un conjunto de datos el objetivo es describir si el mismo está compuesto de dos o más subpoblaciones bien diferenciadas.

Concretamente, el enfoque que utilizaremos en este capítulo es intentar determinar si una muestra procede de dos o más distribuciones normales. Las mixturas son en realidad un caso particular de un conjunto de modelos denominados de variable latente o ausente.

5.1 Modelos de mixturas finitos

En la mayoría de textos, una densidad de mixtura queda representada como una suma ponderada de funciones de densidad independientes (o una combinación convexa):

$$\sum_{j=1}^{k} p_j f_j(x),; p_j \geq 0; \sum_{j=1}^{k} p_j = 1 \tag{5.1}$$

Donde k es el número de subpoblaciones. En las situaciones más simples, las distribuciones f_j se conocen y lo que interesa es estimar las cantidades p_j, o en las probabilidades de que dado un elemento de la muestra, pertenezca a una subpoblación u otra. Generalmente, las distribuciones anteriores pertenecen a una distribución paramétrica, por lo que hay que incluir θ:

$$\sum_{j=1}^{k} p_j f(x|\theta_j) \tag{5.2}$$

Dependiendo de la situación los objetivos de las mixturas pueden ser varios: puede ser estimar la pertenencia a los grupos de las observaciones z (clustering), para proporcionar estimadores de los parámetros de las distribuciones asociadas o incluso estimar el número de poblaciones subyacentes en el modelo.

En cualquier caso, consideremos una muestra aleatoria simple $x = (x_1, \ldots, x_n)$ procedente de un modelo acorde a 5.1. Consideremos la función de verosimilitud asociada:

$$l(\theta, p|x) = \prod_{i=1}^{n} \sum_{j=1}^{k} p_j f(x_i|\theta_j) \tag{5.3}$$

A continuación empezamos el camino para intentar enfocar este problema a un marco de simulación MCMC. Para cualquier distribución a priori $\pi(\theta, p)$, la distribución a posteriori de (θ, p) está disponible hasta constante multiplicativa (como de costumbre):

$$\pi(\theta, p|x) \propto \left[\prod_{i=1}^{n} \sum_{j=1}^{k} p_j f(x_i|\theta_j) \right] \pi(\theta, p) \tag{5.4}$$

Para entender mejor cómo vamos a montar el modelo para la simulación, consideremos el enfoque de variable latente anteriormente mencionado. Para cada x_i, diremos que tiene asociada una variable latente z_i que indica la pertenencia de esa observación a una determinada subpoblación i. Una vez entendido esto, comenzamos definiendo algunas distribuciones condicionadas de manera adecuada:

$$z_i|p \sim M_k(p_1, ..., p_k) \tag{5.5}$$

En primer lugar, la variable latente z_i condicionada a p se distribuirá según una distribución multinomial de parámetros definidos por p.

$$x_i|z_i, \theta \sim f(.|\theta_{z_i}) \tag{5.6}$$

Las observaciones condicionadas a la subpoblación que pertenecen y a los parámetros de la subpoblación que pertenecen se distribuyen según f. En nuestro caso ésta función de densidad será la de una normal, pero el modelo es extensible a cualquier otra distribución (con mayor o menor dificultad). Con esta estructura, podemos definir de nuevo la función de verosimilitud:

$$l(\theta, p|x, z) = \prod_{i=1}^{n} p_{z_i} f(x_i|\theta_{z_i}) \tag{5.7}$$

Donde $z = (z_1, ..., z_n)$.Por tanto la función a posteriori definida anteriormente pasaría a tener la siguiente forma, sustituyendo:

$$\pi(\theta, p|x, z) \propto \left[\prod_{i=1}^{n} p_{z_i} f(x_i|\theta_{z_i}) \right] \pi(\theta, p) \tag{5.8}$$

Ahora vamos a realizar una expansión conveniente para la expresión del modelo. Denotemos por $Z = 1, ..., k^n$, el conjunto de los k^n valores posibles del vector z. Z se puede descomponer trivialmente usando unión de conjuntos $Z = \cup_{j=1}^{\tau} Z_j$. Siguiendo con este razonamiento, dado un vector de número de asignaciones a cada grupo $(n_1, ..., n_k)$, definimos una serie de conjuntos de partición:

$$Z_j = \left\{ z : \sum_{i=1}^{n} I_{z_i=1}, ..., \sum_{i=1}^{n} I_{z_i=k} \right\} \tag{5.9}$$

Es un poco enrevesado seguir lo siguiente: lo anterior representa todas las posibles asignaciones dado un número de asignaciones $(n_1, ..., n_k)$. Nombramos a cada uno de estos conjuntos de partición $j = j(n_1, ..., n_k)$, usando el órden lexicográfico. Utilizando esto, la distribución a posteriori puede escribirse de forma cerrada:

$$\pi(\theta, p|x) = \sum_{z \in Z} \pi(\theta, p|x, z) = \sum_{i=1}^{\tau} \sum z \in Z_i w(z) \pi(\theta, p|x, z) \tag{5.10}$$

Hemos enrevesado tanto la función a posteriori con dos finalidades: la primera encontrar una expresión cerrada para la función a posteriori, y lo segundo es para comprobar que $w(z)$ representa la probabilidad marginal posterior de una asignación a z condicionada a x. Es por ello que un estimador Bayes de la distribución anterior es inmediato:

$$E[\theta, p|x] = \sum_{i=1}^{\tau} \sum_{z \in Z_i} w(z) E[\theta, p|x, z] \tag{5.11}$$

Esta descomposición tan poco natural que hemos desarrollado tiene bastante utilidad en otros métodos que preceden a las simulaciones MCMC que vamos a desarrollar a partir de ahora, concretamente al algoritmo EM.

5.2 Usando MCMC

Llegados a este punto, nos planteamos cómo aplicar lo que hemos visto a las mixturas. La distribución totalmente condicionada de $z|x$ está disponible salvo constante multiplicativa:

$$\pi(z|x, \theta, p) \propto \prod_{i=1}^{n} p_{z_i} f(x_i|\theta_{z_i}) \tag{5.12}$$

A continuación falta determinar una serie de distribuciones conjugadas para las posteriori. Supongamos que p y θ son independientes a priori, entonces, dado z los vectores p y x son independientes:

$$\pi(p|z,x) \propto \pi(p) f(z|p) f(x|z) \propto \pi(p|z) \tag{5.13}$$

De manera similar, θ es también independiente a posteriori de $p|z$ y x, con densidad $\pi(\theta|z,x)$. Aquí ya podemos aplicar el muestreo de Gibbs, simulando por una parte z condicionado sobre (p, θ) y sobre los datos (y al revés). De esta manera, nuestro algoritmo tendría que hacer lo siguiente:

1. Inicio: Escoger $p^{(0)}$ y $\theta^{(0)}$ cualesquiera.
2. $(t++)$ Para cada elemento de la muestra, generar $z_i^{(t)}$ tal que $P(z_i = j|\theta, p) \propto p_j^{(t-1)} f(x_i|\theta_j^{(t-1)})$.
3. Generar $p^{(t)}$ según $\pi(p|z^{(t)})$
4. Generar $\theta^{(t)}$ según $\pi(\theta|z^{(t)}, x)$

Para simular p, lo suyo es escoger una distribución conjugada en z. La complexidad de la simulación de los θ_j dependerá de la f con la que estemos trabajando (en nuestro caso, como son normales y existen distribuciones conjugadas para ambos parámetros no hay problema). Para lo primero que hemos mencionado, los z_i se distribuyen como hemos dicho anteriormente según una multinomial $M(p_1, ..., p_k)$, que permite encontrar fácilmente una distribución conjugada en z. De esta manera, la distribución a priori de $p \sim D(\gamma_1, ..., \gamma_k)$ será una Dirichlet, con densidad:

$$\frac{\Gamma(\gamma_1 + ... + \gamma_k)}{\Gamma(\gamma_1)...\Gamma(\gamma_k)} p_1^{\gamma_1} ... p_k^{\gamma_k} \tag{5.14}$$

De esta manera, la distribución $p|z$ será Dirichlet también:

$$p|z \sim D(n_1 + \gamma_1, ..., n_k + \gamma_k) \tag{5.15}$$

Ejercicio 5.1 R no trae una función de base para simular variables aleatorias según una distribución Dirichlet. Implementar una es fácil a través de valores aleatorios procedentes de distribuciones gamma o beta. Implementemos uno de manera rápida.

```
#!/usr/local/env RScript

rdirichlet <- function(n=1,params=c(1,1)){
  k = length(params)
  array = matrix(NA, nrow = n, ncol = k)
  for(i in 1:n){
    support = rgamma(k,shape=params)
    array[i,] = support/sum(support)
  }
  return(array)
}
```

Para las mixturas normales, las $\mu_j|\sigma_j,z,x$ son independientes, con distribución:

$$\mu_j|\sigma_j,z,x \sim N\left(\varepsilon_j(z), \frac{\sigma_j^2}{n_j+l_j}\right) \tag{5.16}$$

donde:

$$\varepsilon_j(z) = \frac{l_j\varepsilon_j + n_j\bar{x}_j(z)}{l_j+n_j} \tag{5.17}$$

De manera similar:

$$\sigma_j^2|x,z \sim IG(0{,}5(v_j+n_j), 0{,}5s_j(z)) \tag{5.18}$$

donde:

$$s_j(z) = s_j^2 + n_j s_j^2(z) + \frac{l_j n_j}{l_j+n_j}(\varepsilon_j - \bar{x}_j(z))^2 \tag{5.19}$$

Después de tanta teoría, va siendo hora de que implementemos nuestro muestreo de Gibbs orientado a mixturas gaussianas con k componentes. En el código siguiente, asumimos que $l_j = 1$ $v_j = 20$:

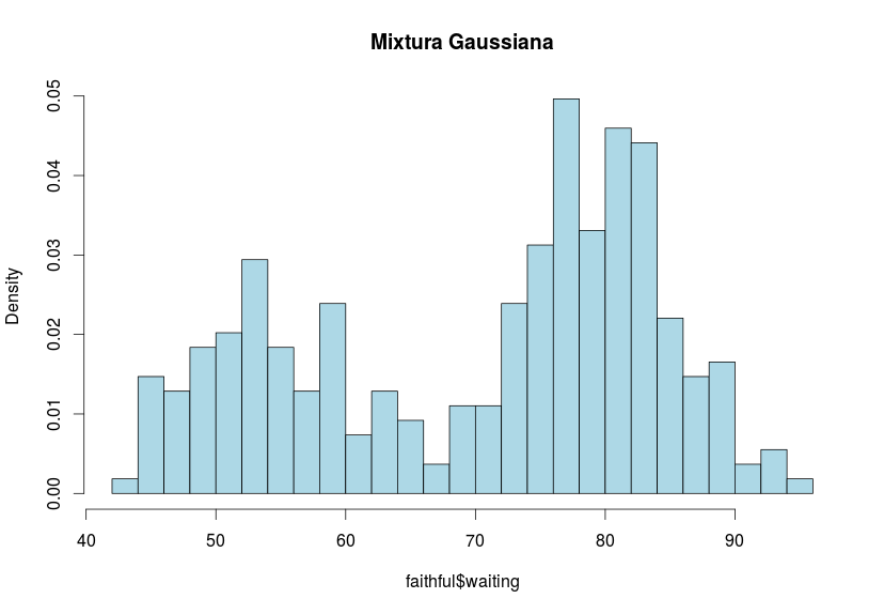
Ejercicio 5.2

```
#!/usr/bin/env RScript
# Gibbs Sampler (Gaussian mixtures)

gibbsMixture <- function(data,k,max_iter=1000){
  n = length(data)
  mu = mean(data)
  sig = var(data)

  # Preallocating data
  z = rep(NA,n)
  group_size = rep(NA, k)
  group_x = rep(NA, k)
  group_sum = rep(NA, k)

  mu_mat = sig_mat = pj_mat = matrix(NA, nrow=max_iter, ncol=k)
  mu_mat[1,] = rep(mu,k)
```

```
  pj_mat[1,] = rep(1,k)/k
  sig_mat[1,] = rep(sig,k)

  ## Possible chunk for likelihood

  #######

  ### Main loop

  for(i in 2:max_iter){
    # z estimation
    for(j in 1:n){
      p = pj_mat[i-1,]*dnorm(data[j],mu_mat[i-1,],sqrt(sig_mat[i-1,]))
      z[j] = sample(1:k, size = 1, prob = p)
    }

    # rest of parameters

    for(j in 1:k){
      group_size[j] = length(which(z == j))
      group_x[j] = sum(as.numeric(z == j)*data)
      group_sum[j] = sum(as.numeric(z==j)*(data-group_x[j]/group_size[j])^2)
    }

    mu_mat[i,] = rnorm(k, (mean(data) + group_x)/(group_size + 1),
     sqrt(sig_mat[i-1,]/group_size + 1))
    sig_mat[i,] = 1/rgamma(k, .5*(20+group_size),
                          var(data) + .5* group_sum +
                          .5* group_size/(group_size + 1)*(mean(data) -
                          group_x/group_size)^2)

    pj_mat[i,] = rdirichlet(params = group_size + 0.5)



  }
  class = setClass("MCMC mixture",
   slots=c(mu="matrix", sig="matrix", p="matrix", group="numeric"))
  res = class(mu = mu_mat, sig = sig_mat, p = pj_mat, group = z)
  return(res)
}
```

Ejercicio 5.3 Usamos un marco de datos clásico como faithful, incluído en R para poner a prueba nuestro algoritmo. En primer lugar representamos los datos para hacernos una idea visual de si un modelo de mixturas es adecuado.

```
data(faithful)

hist(faithful$waiting, breaks = 20, col="lightblue",
     freq=F, main="Mixtura Gaussiana")
```

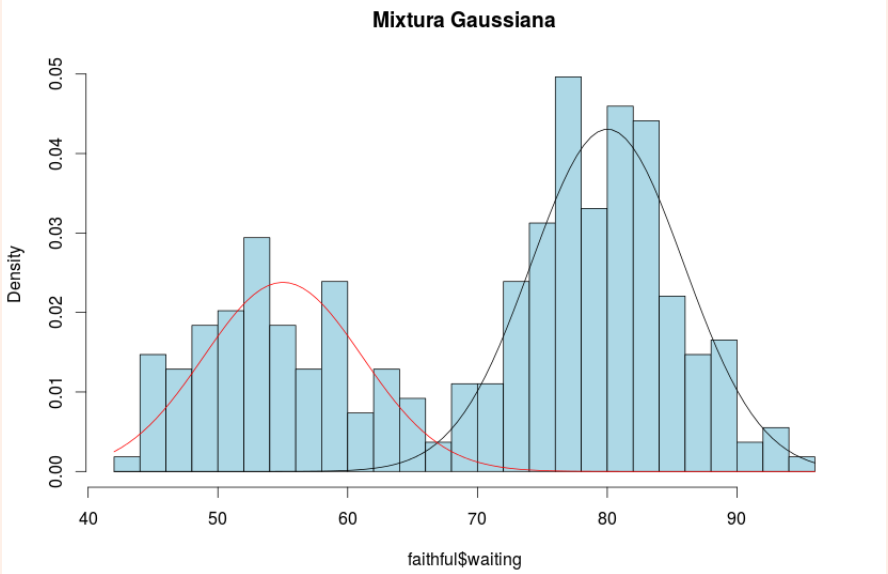


Aparentemente parece que podemos modelar dicho conjunto de datos según una mixtura gaussiana de $k = 2$ componentes. Aplicamos nuestro algoritmo y tomamos medias:

```
set.seed(93)
mixture = gibbsMixture(faithful$waiting,k = 2,max_iter = 1000)
(mu = apply(mixture@mu, 2, mean))
[1] 55.11955 80.08556
(sig = apply(mixture@sig, 2, mean))
[1] 38.28160 33.79188
(p = apply(mixture@p, 2, mean))
[1] 0.367852 0.632148
```

Finalmente, podemos superponer sobre el histograma anterior la densidad de nuestra mixtura:

```
curve(p[1]*dnorm(x,mu[1],sqrt(sig[1])),add=T, lwd=2)
curve(p[2]*dnorm(x,mu[2],sqrt(sig[2])),add=T, col="red", lwd=2)
```



Los resultados anteriores pueden llevar a una falsa sobrevaloración del modelo estadístico entre manos. Recordemos que el muestreador de Gibbs está condicionado a los valores iniciales que escojamos para la cadena. El condicionamiento sobre z implica que las cadenas y por tanto tanto θ como p son incapaces de realizar cambios drásticos sobre sí mismos y sobre las asignaciones en la siguiente iteración. Otras técnicas anteriores (como el algoritmo EM) también son sensibles a no realizar cambios drásticos sobre asignaciones y por tanto son igual de frágiles con respecto a este sentido.

Una posible solución es utilizar el muestreo de Gibss en conjunto con alguna otra técnica MCMC, como Metropolis-Hastings. Podríamos usar en este sentido una probabilidad de aceptación con una adecuada distribución propuesta del tipo:

$$\frac{\pi(\theta', p'|x)}{\pi(\theta, p|x)} \frac{q(\theta, p|\theta', p')}{q(\theta', p'|\theta, p)} \tag{5.20}$$

5.3 Dificultad en reasignación

Como se ha visto en la sección anterior, al muestreo de Gibbs le cuesta realizar cambios drásticos sobre reasignaciones en z. Una característica curiosa en los modelos de mixturas es que carece de un orden, es decir, dada una mixtura compuesta por dos densidades, es incorrecto decir que una de las dos es el primer componente de la misma o el segundo. De manera más técnica decimos que los parámetros θ_j no son marginalmente identificables.

Por si fuera poco, en una mixtura de k componentes se puede probar que el número de modas en la verosimilitud es del orden de $O(k!)$. Esto es fácilmente comprobable ya que si (θ, p) es un máximo local en la verosimilitud, entonces una permutación de estos parámetros también sigue siéndolo. Es más, si usamos una distribución a priori sobre (θ, p) la cual es invariable bajo permutación de los índices, todas las distribuciones a posteriori marginales son idénticas, lo cual implicaría que los estimadores Bayes son los mismos.

Este problema sobre reasignaciones tiene distintas soluciones. Por un lado, una solución

sencilla podría llevarse a cabo imponiendo una restricción de orden en los parámetros (por ejemplo, ordenando las medias). Esto se reduce en su totalidad a truncar la distribución a priori:

$$\pi(\theta, p) I_{\mu_1 \leq \ldots \leq \mu_k} \tag{5.21}$$

Con esta reducción del espacio paramétrico pueden ocurrir sucesos inesperados, ya que las distribuciones a posteriori no tienen por qué respetar su topología, de tal manera que cuando ponemos simulaciones a funcionar los estimadores no tienen por qué acabar en una de las $k!$ mencionadas anteriormente sino en una zona de silla de baja probabilidad. No obstante, si bien esta restricción puede suponer un cambio de rendimiento en las simulaciones, éstas no tienen por qué realizarse durante la misma sino después. En este sentido, si la restricción es sobre el orden en las medias, una vez que la simulación ha acabado, los componentes se pueden reasignar de acuerdo con este orden.

Esto se puede realizar de la siguiente manera: dada una muestra de tamaño M de una simulación MCMC, definimos el estimador máximo a posteriori como:

$$i^* = argmax_{i=1,\ldots,M}\, \pi\{(\theta, p)^{(i)} | x\} \tag{5.22}$$

En palabras, es el valor simulado que maximiza la función de densidad a posteriori. Esta, como ya sabemos, no es necesario que incluya la constante de integración. Sin embargo, es bastante problable que este valor se encuentre en la vecindad de una de las $k!$ posibles modas. Usaremos este valor óptimo (MAP) como pivote, en el sentido de que ordenaremos el resto de las otras iteraciones con respecto a dicha moda.

En lugar de escoger este reorden según una distancia euclídea en el espacio paramétrico, la definimos en el espacio probabilístico de las asignacioens. Denotemos por G_k el conjunto de las k posibles permutaciones y $\tau \in G_k$. Minimizamos en τ una distancia de entropía:

$$h(i, \tau) = \sum_{t=1}^{n} \sum_{j=1}^{k} P(z_t = j | \theta^{(i*)}, p^{(i*)}) \times \log \left(\frac{P(z_j = k | \theta^{(i*)}, p^{(i*)})}{P(z_t = j | \tau(\theta^{(i)}, p^{(i)}))} \right) \tag{5.23}$$

En definitiva, podemos definir nuestro algoritmo de reordenamiento pivotal como:

- En todas las iteraciones, calcular $\tau_i = argmin\, h(i, \tau)$
- $(\theta^{(i)}, p^{(i)}) = \tau_i\{(\theta^{(i)}, p^{(i)})\}$

Con este reordenamiento, en la mayoría de iteraciones las asignaciones quedan reasignadas a la misma moda, y por tanto eliminamos el problema de la identificación. Después de este reordenamiento, los estimadores Monte-Carlo siguen siendo los habituales. Realicemos la implementación de este reordenamiento en R:

Ejercicio 5.4 Suponemos que tenemos una salida de la función gibbsMixture, la cual usamos en esta función:

```
pivotalReor <- function(gibbsMix){
  mu = gibbsMix@mu
  sig = gibbsMix@sig
  p = gibbsMix@p
  logpost = gibbsMix@logpost
  n = gibbsMix@n
  k = gibbsMix@k
  data = gibbsMix@data
```

```
  max_iter = gibbsMix@max_iter

  map_indices = order(logpost, decreasing = T)[1]
  map = list(mu = mu[map_indices,], sig= sig[map_indices,],
             p = p[map_indices,])

  lili = matrix(NA, n, k)
  allocation = matrix(NA, n, k)

  for(t in 1:n){
    lili[t,] = map$p*dnorm(data[t], mean = map$mu, sd=sqrt(map$sig))
    lili[t,] = lili[t,]/sum(lili[t,])
  }

  ordered_mu = matrix(NA, ncol=k, nrow=max_iter)
  ordered_sig = matrix(NA, ncol=k, nrow=max_iter)
  ordered_p = matrix(NA, ncol=k, nrow=max_iter)

  require(combinat)

  perma = permn(k)

  for(t in 1:1000){
    entropies = rep(0, factorial(k))
    for(j in 1:n){
      allocation[j,] = p[t,]*dnorm(data[j], mean=mu[t,], sd=sqrt(sig[t,]))
      allocation[j,] = allocation[j,]/sum(allocation[j,])
      for(i in 1:factorial(k)){
        entropies[i] = entropies[i] + sum(lili[j,]*log(allocation[k,perma[[i]]]))
      }
    }
    best_ordering = order(entropies, decreasing=T)[1]
    ordered_mu[t,] = mu[t,perma[[best_ordering]]]
    ordered_sig[t,] = sig[t,perma[[best_ordering]]]
    ordered_p[t,] = p[t,perma[[best_ordering]]]
  }

  res = list(mu=ordered_mu, sig=ordered_sig, p=ordered_p)
  return(res)

}
```

Ejercicio 5.5 Pongamos en práctica el reordenamiento pivotal con el marco de datos que hemos estado utilizando antes:

```
pivote = pivotalReor(mixture)
(mu_ordenado = apply(pivote$mu,2,mean))
[1] 80.08642 55.11868
(sig_ordenado = apply(pivote$sig,2,mean))
[1] 33.79117 38.28231
(p_ordenado = apply(pivote$p,2,mean))
[1] 0.6320439 0.3679561
```

Con estos resultados, las mixturas sí son perfectamente identificables, y podríamos decir que nuestro marco de datos se compone de dos subpoblaciones, constituyendo una mixtura $0{,}632N(80{,}08, 33{,}8) + 0{,}368N(55{,}12, 38{,}28)$. ■

5.4 Determinando el número de subpoblaciones

Hasta ahora hemos supuesto que conocemos el número k de las subpoblaciones y toda la inferencia se ha basado en estimar parámetros para estas poblaciones. Existe un enfoque a las mixturas, semiparamétrico en el cual k no representa más que el grado de aproximación del modelo a los datos. No obstante, este enfoque queda fuera de la intención del trabajo y adoptaremos un estilo más cercano a la selección de modelos. Cuando se trata de mixturas, generalmente se tiene un número pequeño $(4, 6...)$ de modelos que probar y la inferencia se basa en escoger uno de ellos.

Generalmente, para escoger entre estos modelos se utiliza un estadístico denominado factor de Bayes, que no es más que un cociente de verosimilitudes. Una vez hemos aceptado que vamos a probar varios modelos, tendríamos que aproximar los factores de Bayes a través de la simulación MCMC pertinente. Consideremos nuestra verosimilitud:

$$f_j(x|\lambda_j) = \prod_{i=1}^{n} \sum_{j=1}^{J} p_j f(x_i|\theta_j) \tag{5.24}$$

donde por resumir $\lambda_j = (\theta, p)$. La técnica que utilizaremos aquí para estimar el número de subpoblaciones se propuso por Chib (1995), que aunque es susceptible a fallar por el problema de reasignación se propone una corrección. Este método está basado en la distribución marginal:

$$m_j(x) = \frac{f_j(x|\lambda_j)\pi_j(\lambda_j)}{\pi_j(\lambda_j|x)} \tag{5.25}$$

Si una buena aproximación a λ_j se ha obtenido e igualmente a $\pi_j(\lambda_j|x)$, el estimador quedaría:

$$\hat{m}_j(x) = \frac{f_j(x|\lambda_j^*)\pi_j(\lambda_j^*)}{\hat{\pi}_j(\lambda_j|x)} \tag{5.26}$$

En nuestro caso, λ_j^* puede ser aproximado bien por lo denominado en el apartado anterior como MAP. El estimador de $\pi(\lambda_j|x)$ que tiene en cuenta el problema de reasignación es el siguiente:

$$\pi(\lambda_j|x) = \frac{1}{TJ!} \sum_{\sigma \in G_j} \sum_{t=1}^{T} \pi_j\left(\sigma(\lambda_j^*)|x, z^{(t)}\right) \tag{5.27}$$

donde G_j indica el conjutno de todas las permutaciones posibles de $\{1,...,J\}$ y $\sigma(\lambda_j^*)$ indica la transformación de λ_j^* donde todos los componentes han sido permutados de acuerdo con σ.

Ejercicio 5.6 Implementaremos este método de estimación de componentes con R, utilizando la salida de la función `gibbsMixture`, o un objeto de tipo `MCMC Mixture` cualquiera.

```
chibComp <- function(gibbsMix){

  logpost = gibbsMix@logpost
  k = gibbsMix@k
  mu = gibbsMix@mu
  sig = gibbsMix@sig
  p = gibbsMix@p
  max_iter = gibbsMix@max_iter
  data = gibbsMix@data

  group_size = gibbsMix@group_size
  group_x = gibbsMix@group_x
  group_x2 = gibbsMix@group_x2
  group_sum = gibbsMix@group_sum

  meanp = mean(data)
  varp = var(data)

  require(bayess)

  mix = list(k=k, mu=mu[1,],sig=sig[1,],p=p[1,])
  simu = gibbs(max_iter,data,mix)
  chib = simu$deno
  print(chib)
  lolik = simu$lolik

  lopos = order(logpost)[max_iter]
  part1 = lolik[lopos]
  part2 = sum(dnorm(mu[lopos,],
                    mean=meanp, sd=sig[lopos,],log=T)+
                dgamma(1/sig[lopos,],10,varp,log=T)-
                2*log(sig[lopos,]))+
    sum((rep(0.5,k)-1)*log(p[lopos,]))+
    lgamma(sum(rep(0.5,k)))-sum(lgamma(rep(0.5,k)))
  def = part1 + part2 - log(chib)
  return(def)

}
```

La anterior función hace uso del paquete `bayess`, por eficiencia a la hora de calcular el denominador del factor deseado.

Podríamos aplicar esta función a nuestro marco de datos `faithful`.

```
set.seed(93)
chib = numeric(3)
data = faithful$waiting

for(i in 2:4){
  model = gibbsMixture(data,k=i)
  chib[i-1] = chibComp(model)
}
> chib
[1] -861.3397 -685.4013 -468.3826
```

■

6. Análisis de imagen

En este capítulo analizaremos otra serie de problemas, orientados al tratamiento de imágenes para los cuales los métodos MCMC son una resolución natural. De hecho, muchos de los algoritmos estudiados en otros capítulos, como el muestreador de Gibbs tienen su origen en el mundo físico y en el análisis de imagen.

Nos centraremos en una problemática particular: la segmentación de imágenes. Supondremos que una imagen determinada tiene un ruido que queremos eliminar para obtener la imagen original. Esto tiene aplicaciones como procesamiento de imágenes para un posterior procedimiento de clasificación o para reconstruír imágenes de satélites con obstáculos (nubes) entre otras. Dentro de esta parte veremos dos modemos para reconstruír la imagen, ambos basados en el muestreador de Gibbs pero dependiendo del número de colores que tenga nuestra imagen. Esto tiene aplicaciones inmediatas en reconocimiento de objetos y otras técnicas similares.

6.1 Reconstrucción de imágenes

Debemos empezar a preguntarnos en primer lugar qué es una imagen desde el punto de vista matemático. Podemos interpretar una imagen digital en píxeles como una cuadrícula. En este sentido, cada píxel (cada ítem de la cuadrícula) puede ser considerado como un elemento aleatorio. Cada elemento de la cuadrícula está indexado por la localización geográfica de cada píxel. Por tanto, cada píxel está relacionado por la proximidad geográfica con otros píxeles. En general y matemáticamente hablando, una cuadrícula es una entidad multidimensional en la que puede ser definida una relación de proximidad.

Por facilidad, podemos pensar en una cuadrícula como en una matriz $n \times m$, donde cada elemento (i, j) tiene de vecinos a los elementos $(i-1, j), (i+1, j), (i, j-1), (i, j+1)$. Para trabajar con estas estructuras, es natural expandir el concepto de cadenas de Markov multidimensionalmente. El primer paso para realizar esta generalización es definir un vecindario de manera formal: Dada una cuadrícula I, con píxeles $i \in I$ de una imagen, una relación de vecindad en I se define como $i \sim j$, significando que i es vecino de j. Si definimos un vector x indexado en la cuadrícula $x = (x_i)_{i \in I}$, significaría que los componentes x_i y x_j están correlados si i, j son vecinos ($i \sim j$). Lógicamente esta relación es simétrica: si i es vecino de j, el recíproco también es cierto.

6.1.1 Campos aleatorios de Markov

Aquí expandimos el concepto de cadena de Markov a varias dimensiones de manera muy breve. Definimos un campo aleatorio sobre I es una estructura aleatoria indexada por I con una colección de variables aleatorias $\{x_i \, i \in I\}$, donde cada x_i toma valores en un conjunto finito χ. En nuestro caso, los x_i serán variables dependientes con respecto a la relación de vecindad de la cuadrícula.

Notemos por $n(i)$ el conjunto de vecinos de $i \in I$. De esta manera $x_A = \{x_i \, i \in A\}$ denota el subconjunto de x para los índices en el subconjunto $A \in I$, y por tanto $x_{n(i)}$ es el conjunto de valores tomado por los vecinos de i.

Decimos que un campo aleatorio es de Markov (MRF) si la distribución condicional de cualquier píxel, dados cualesquiera otros, solo depende de los valores de los vecinos. Es decir:

$$\pi(x_i|x_{-i}) = \pi(x_i|x_{n(i)}) \tag{6.1}$$

donde x_{-i} denota a todos los píxeles de x salvando el i-ésimo.

Con estas definiciones, tenemos más que suficiente para explicar dos campos aleatorios de Markov específicos para el tratamiento de imágenes. Uno de ellos el modelo de Ising, para imágenes binarias y la extensión a más colores, el modelo de Potts.

6.1.2 Modelo de Ising

Supongamos que tenemos una imagen en la cual solamente hay presente dos colores (blanco y negro, generalmente). Este caso, x es binario, y por convención nos referimos a un píxel como frente si $x_i = 1$, o fondo si $x_i = 0$. Parece bastante natural decir que la distribución de un píxel es Bernoulli, con su correspondiente parámetro dependiendo del número de vecinos de un color u otro que tenga. Por ejemplo, diremos que la distribución condicional de un píxel solo es dependiente del número de vecinos con color negro, usando una función de enlace logit:

$$\pi(x_i = j|x_{-i}) \propto exp\,(\beta n_{i,j}) \tag{6.2}$$

donde $n_{i,j} = \sum_{l \in n(i)} I_{x_l = j}$ es el número de vecinos de i con color j. La distribución totalmente condicionada en el modelo de Ising queda definida de esta manera:

$$\pi\,(x_i = 1|x_{-i}) = \frac{exp(\beta n_{i,1})}{exp(\beta n_{i,0}) + exp(\beta n_{i,1})} \tag{6.3}$$

Es natural así, que la distribución conjunta sea proporcional a lo siguiente:

$$\pi(x) \propto \left(\beta \sum_{j \sim i} I_{x_j = x_i} \right) \tag{6.4}$$

Nos encontramos a continuación con una dificultad que resolveremos en uno de los siguientes apartados, y es la inferencia sobre β. En esta última distribución nos falta la constante de integración $Z(\beta)$, que salvo para cuadrículas pequeñas es computacionalmente imposible de hallar, y por tanto no podemos hallar una aproximación a la función de verosimilitud. Esta dificultad será resuelta más adelante por el método ABC. No obstante, por ahora consideraremos β conocido y nos centramos en la simulación de x dada una versión con ruido y de la misma imagen.

La ecuación en 6.3 permite actualizaciones píxel a píxel con el muestreador de Gibbs de manera sencilla:

1. Para cada píxel i, generar $x_i^{(0)} \sim B(1/2)$. (Paso de inicialización sin imagen).
2. Resto de iteraciones $(t > 1)$. Construír u, una permutación aleatoria sin repeteción de los elementos de I.
3. Para $1 \leq l \leq |I|$, actualizar $n_{ul,0}^{(t)}$ y $n_{ul,1}^{(t)}$ y generar $x_{ul}^{(t)} \sim B\left(\frac{exp(\beta n_{ul,1})}{exp(\beta n_{ul,0}) + exp(\beta n_{ul,1})}\right)$

Realmente, el paso de la permutación aleatoria no es necesario, pero se utiliza para evitar posibles cuellos de botella en la exploración de la distribución. Una nota empírica, es que cuanto mayor sea nuestro β, la distribución tiene más tendencia a mostrar patrones con un solo color, y por tanto el muestreador de Gibbs encontrará dificultad para cambiar los valores de los píxeles.

Ejercicio 6.1 Implementemos el modelo de Ising en R. En primer lugar necesitamos una función que dada una cuadrícula y una posición (a,b) de la misma, nos devuelva el número de vecinos con un color u otro. Eso es fácilmente implementable:

```
neighbors4 <- function(x,a,b,col){
  n=dim(x)[1]; m = dim(x)[2]
  nei = c(x[a-1,b]==col,x[a,b-1]==col)
  if(a != n){
    nei = c(nei, x[a+1,b]==col)
  }
  if(b != m){
    nei = c(nei, x[a,b+1]==col)
  }
  return(sum(nei))
}
```

A continuación y usando esta función, implementamos el modelo:

```
isingSampler <- function(max_iter, n, m=n, beta){
  x = matrix(sample(c(0,1), n*m, rep=T),ncol=n)

  for(i in 1:max_iter){
    sample_rows = sample(1:n)
    sample_cols = sample(1:m)

    for(k in 1:n){
      for(l in 1:m){
        n0 = neighbors4(x, sample_rows[k], sample_cols[l], 0)
        n1 = neighbors4(x, sample_rows[k], sample_cols[l], 1)
        x[sample_rows[k], sample_cols[l]] = sample(c(0,1), 1, prob = exp(beta*c(n0,n1)))
      }
    }
  }
  return(x)
}
```

No obstante, nos fijamos que esta función implica realizar tres bucles anidados, lo que provoca que sea tremendamente ineficiente. He decidido reimplementar esta función en `C++` por razones de rendimiento. Esta función puede usarse en R sin dificultad usando `Rcpp`. La ganancia en tiempo es de varios órdenes en general con respecto a la original escrita en R.

```
#include <RcppArmadillo.h>
#include <RcppArmadilloExtensions/sample.h>
#include <math.h>       /* exp */

// [[Rcpp::depends(RcppArmadillo)]]
using namespace Rcpp;

// [[Rcpp::export]]
int nei4(NumericMatrix x, int a, int b, int col){
  int n = x.nrow(), m = x.ncol();
  int nei = 0;
  a = a-1;
  b = b-1;
  if(a != 0){
    if(x(a-1,b) == col){nei++;}
    }
  if(b != 0){
    if(x(a,b-1) == col){nei++;}
  }
  if(a != (n-1)){
    if(x(a+1,b) == col){nei++;}
  }
  if(b != (m-1)){
    if(x(a,b+1) == col){nei++;}
  }
  return nei;}

// [[Rcpp::export]]
NumericMatrix isingSampler(NumericMatrix x, int max_iter, double beta){
  int n = x.nrow(), m = x.ncol();
  int i;
  int k;
  int l;

  NumericVector rows(n);
  rows = Rcpp::seq(1,n);
  NumericVector cols(m);
  cols = Rcpp::seq(1,m);

  NumericVector pos(2);
  pos = Rcpp::seq(0,1);

  NumericVector permut_rows(n);
  NumericVector permut_cols(m);

  NumericVector prob(2, 0.5);

  int n0;
  int n1;
```

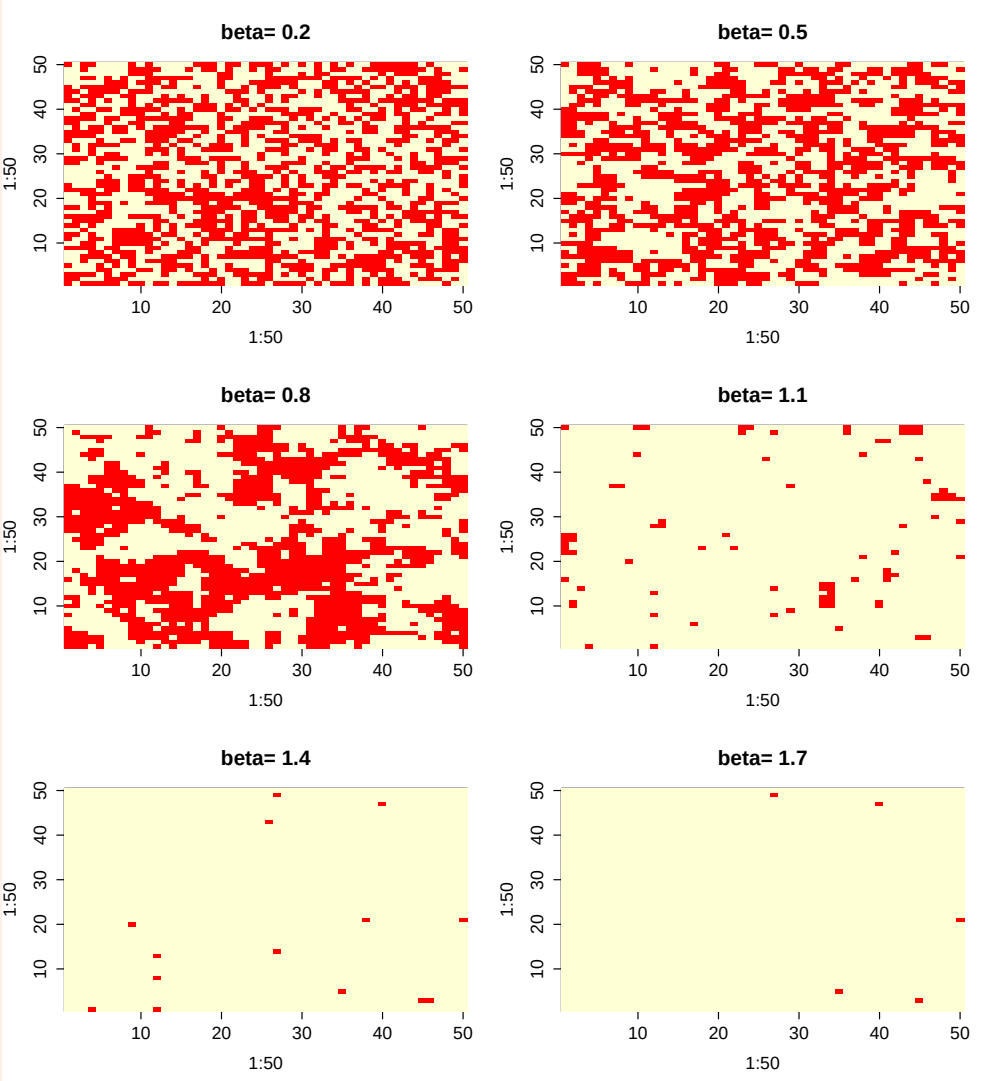
```
  int pos1;
  int pos2;


  double res;


  for(i = 0; i < max_iter; i++){
    permut_rows = RcppArmadillo::sample(rows,n,0);
    permut_cols = RcppArmadillo::sample(cols,m,0);


    for(k = 1; k <= n; k++){
      for(l = 1; l <= m; l++){
        n0 = nei4(x, permut_rows[k-1], permut_cols[l-1], 0);
        n1 = nei4(x, permut_rows[k-1], permut_cols[l-1], 1);
        prob[0] = exp(beta*n0);
        prob[1] = exp(beta*n1);


        pos1 = permut_rows[k-1]-1;
        pos2 = permut_cols[l-1]-1;

        double suma = prob[0] + prob[1];
        prob[0] = prob[0]/suma;
        prob[1] = prob[1]/suma;

        res = RcppArmadillo::sample(pos, 1, 0, prob)[0];
        //res = Rcpp::Function(sample(pos,1,0,prob));
        Rcpp::Function(gc());


        x(pos1, pos2) = res;

      }
    }
    Rcpp::Function(gc());

  }

  return x;

}
```

Un asunto de interés es comprobar cómo se comporta el algoritmo dependiendo del valor de β escogido. Podemos realizar una prueba empírica rápida:

```
betas = seq(0.2, 1.8, by = 0.3)
```

```
par(mfrow=c(3,2))
set.seed(39)
x = matrix(sample(c(0,1),50*50,rep=T), nrow=50)

for(beta in betas){
  set.seed(93)
  y = isingSampler(x, 1000, beta)
  image(y, main=paste("beta=",beta))
}
```

Que produce el siguiente resultado:



Como se puede comprobar, cuanto mayor es β, el modelo tiende a concentrarse bajo distribuciones de un solo color, y por tanto más homogénea la imagen.

6.1.3 Modelo de Potts

El modelo de Potts es una generalización natural al modelo de Ising cuando la imagen tiene más de dos colores, por ejemplo G. Esta vez notamos por $n_{i,g}$ al número de vecinos de i con color g, es decir, $n_{i,g} = \sum_{j\sim i} I_{x_j=g}$. La distribución totalmente condicionada de x_i se escoge de manera idéntica como:

$$\pi(x_i = g|x_{-i}) \propto exp(\beta n_{i,g}) \tag{6.5}$$

La densidad conjunta del modelo de Potts, como ya esperamos tiene la forma:

$$\pi(x) \propto exp\left(\beta \sum_{j\sim i} I_{x_j=x_i}\right) \tag{6.6}$$

Aquí nos encontramos con el mismo problema que en la sección anterior, y es que no conocemos el valor de β, y por tanto no podemos calcular la función de verosimilitud. Por otra parte, con un β grande, tal y como pasaba antes, es difícil que un valor se actualice y se reduce la velocidad de convergencia. Se propone una modificación basada en pasos Metropolis-Hastings que fuerza actualizaciones en cada paso.

1. Inicialización: Para cada $i \in I$, generar $x_i^{(0)} \sim UD(1,...,G)$
2. Iteración $t > 1$. Generar u, una permutación aleatoria de los elementos de I.
3. Para l desde 1 hasta $|I|$:
   generar $x_{u,l} \sim UD\left(1,2,...,x_{u,l}^{(t-1)} - 1, x_{u,l}^{(t-1)} + 1,...,G\right)$
   generar número de vecinos $n_{ul,g}^{(t)}$ y
   generar probabilidad de aceptación $p_l = max\left\{exp(\beta n_{ul,x_{ul}})/exp(\beta n_{ul,x_{ul}})^{(t)}, 1\right\}$
   si se acepta, sustituir x_{ul} por $x_{ul}^{(t)}$

Ejercicio 6.2 De nuevo seguimos con la misma mecánica aplicada en el modelo anterior. Una implementación ineficiente del algoritmo en R podría ser la siguiente:

```
pottsSampler <- function(x, num_col=2, max_iter=1000, beta){

  n = dim(x)[1]; m = dim(x)[2]

  for(i in 1:max_iter){
    permut = sample(1:(n*m))
    cat("Iteracion", i, "\n")

    for(k in 1:(n*m)){
      xcur = x[permut[k]]
      a = (permut[k]-1)%%n + 1
      b = (permut[k]-1)%/%n + 1
      xtilde = sample((1:num_col)[-xcur],1)
      prob = beta*(nei4(x,a,b,xtilde)-nei4(x,a,b,xcur))
      if(log(runif(1))<prob){
        x[permut[k]] = xtilde
      }
    }
```

```
  }
  return(x)
}
```

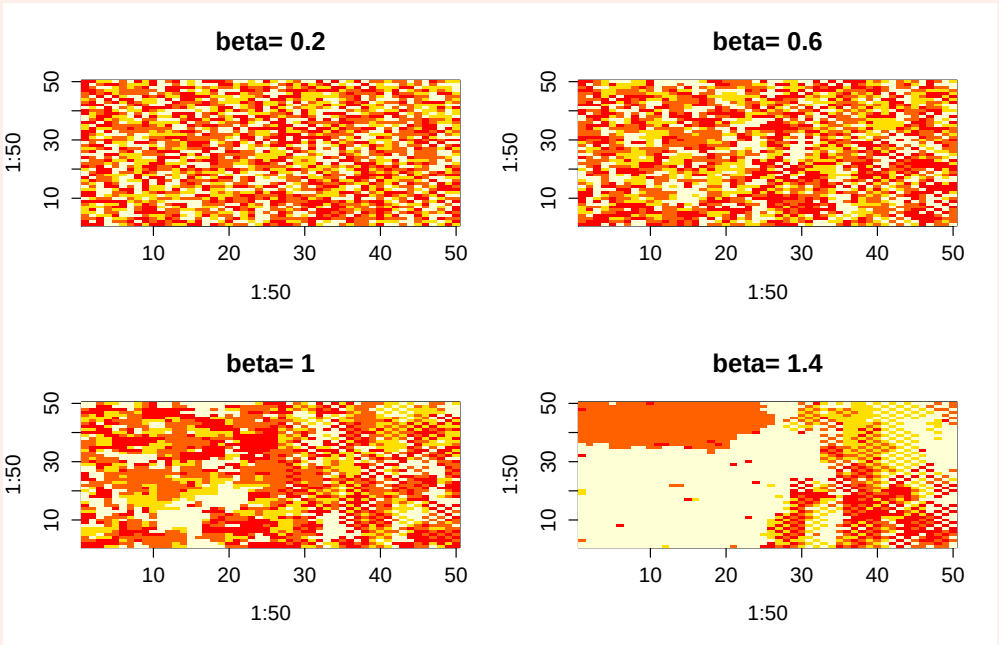
Y una varios órdenes más eficiente, en C++.

```
#include <RcppArmadillo.h>
#include <RcppArmadilloExtensions/sample.h>
#include <math.h>

// [[Rcpp::depends(RcppArmadillo)]]
using namespace Rcpp;


// [[Rcpp::export]]
NumericMatrix pottsSampler(NumericMatrix x, int num_col, int max_iter, double beta){

int n = x.nrow(), m = x.ncol();
int i;
int xcur;
int k, j;
int a, b;
int xtilde;
double prob;

  NumericVector pixels(n*m);
pixels = Rcpp::seq(1, n*m);

for(i = 0; i < max_iter; i++){

    NumericVector permut(n*m);
permut = RcppArmadillo::sample(pixels,n*m,1);

for(k = 0; k < (n*m); k++){
      NumericVector colors(num_col);
colors = Rcpp::seq(1, num_col);
xcur = x[permut[k] - 1];
a = remainder((permut[k]-1),n) + 1;
b = (permut[k]-1)/n + 1;
colors.erase(xcur - 1);
xtilde = RcppArmadillo::sample(colors, 1, 0)[0];
prob = beta*(nei4(x, a, b, xtilde) - nei4(x, a, b, xcur));
      double alea = runif(1)[0];
if(log(alea) < prob){
x[permut[k] - 1] = xtilde;
}


}
```

```
}
return x;
}
```

De manera similar a lo que hicimos para el modelo de Ising, podemos comprobar de manera empírica el comportamiendo del modelo teniendo en cuenta el valor de β.

```
betas = seq(0.2, 1.4, by = 0.4)

par(mfrow=c(2,2))
set.seed(39)
x = matrix(sample(1:4,50*50,rep=T), nrow=50)

for(beta in betas){
  set.seed(93)
  y = pottsSampler(x, num_col=4, 1000, beta)
  image(x=1:50, y=1:50,z=y, main=paste("beta=",beta))
}
```



De manera similar, cuanto mayor es el parámetro, más tendencia se tiene hacia imágenes más homogéneas. ■

6.1.4 Sobre la constante de integración

En las secciones anteriores hemos supuesto que β, y que por tanto $Z(\beta)$ eran conocidos. Como cabe esperar, esto no se suele producir en la mayoría de las ocasiones. El manejo de esa constante de integración ha dado lugar a mucha literatura, pues es un problema difícil. Aquí veremos una metodología relativamente sencilla para averiguarla, denominada muestreo por caminos. Esta técnica está basada en una representación de la derivada de dicha constante:

$$\frac{dZ(\beta)}{d\beta} = \sum_x S(x) exp(\beta S(x)) \tag{6.7}$$

Esta derivada puede expresarse de manera conveniente como una esperanza:

$$\frac{dZ(\beta)}{d\beta} = Z(\beta) \sum_x S(x) \frac{exp(\beta S(x)}{Z(\beta)} = Z(\beta) E[S(x)] \tag{6.8}$$

y por tanto, tomando logaritmo antes:

$$\frac{d \log Z(\beta)}{d\beta} = E[S(x)] \tag{6.9}$$

Esta representación permite que podamos expresar el ratio $\frac{Z(\beta_1}{Z(\beta_0)}$ como una integral:

$$\log(Z(\beta_1)/(Z(\beta_0)) = \int_{\beta_0}^{\beta_1} E[S(x)] d\beta \tag{6.10}$$

Esta última ecuación es lo que se denomina identidad del muestreo por caminos. Con esta expresión, podemos recurrir a procedimientos de simulación estándares para su aproximación. La integral puede aproximarse por métodos numéricos, y para un valor de β, $E[S(x)] = f(\beta)$ puede simularse usando el modelo de Potts, por ejemplo. Finalmente aproximamos $f(\beta)$ por una función lineal a trozos para integrar fácilmente.

Ejercicio 6.3 Realicemos una aproximación de $f(\beta)$ usando R. Para un valor determinado de β:

```
pathSampling <- function(x, ncol=2, max_iter = 10^3, beta){
  n = dim(x)[1]; m = dim(x)[2]
  S = 0

  for(i in seq_len(max_iter)){
    if(i%%100==0){
      cat("Iteración ",i,"\n")
    }
    s = 0
    rows = sample(1:n)
    cols = sample(1:m)

    for(k in seq_len(n)){
      for(l in seq_len(m)){
        n0 = nei4(x, rows[k], cols[l], x[rows[k], cols[l]])
        col = sample(1:ncol, 1)
        n1 = nei4(x, rows[k], cols[l], col)
        if(log(runif(1))< (beta*(n1-n0))){
          x[rows[k], cols[l]] = col
          n0 = n1
        }
        s = s+n0
      }
    }
    if(2*i > max_iter){
```

```
      S = S + s
    }
  }
  return(2*S/max_iter)
}
```

Y para generar una secuencia, en intervalos tan pequeños como se quiera:

```
generatefbeta <- function(x, max_iter=10^3, ncol=2){
  search = seq(0.1, 2, by=0.1)
  Z = seq_along(search)
  i = 1
  for(beta in search){
    cat("Beta: ", beta, "\n")
    Z[i] = pathSampling(x, ncol, max_iter, beta)
    i = i+1
  }
  plot(search, Z, main="f(beta) approx.", type="l")
  return(Z)
}
```

El problema en eficiencia es que para imágenes de juguete nuestro algoritmo está bien, pero como venimos acostumbrando en este capítulo, sería buena idea implementar `pathSampling` en C++.

```
double pathSampling(NumericMatrix x, int ncol, int max_iter,
double beta){

int n = x.nrow(), m = x.ncol();
double S = 0.0, s = 0.0;
int i, k, l, n0, n1, col;

NumericVector rows(n);
rows = Rcpp::seq(1, n);

NumericVector cols(n);
cols = Rcpp::seq(1, m);

NumericVector colors(ncol);
colors = Rcpp::seq(1, ncol);


for(i = 0; i < max_iter; i++){
s = 0.0;

rows = RcppArmadillo::sample(rows, n, 0);
cols = RcppArmadillo::sample(cols, m, 0);

for(k = 0; k < n; k++){
```

```
for(l = 0; l < n; l++){
n0 = nei4(x, rows[k], cols[l], x(rows[k]-1, cols[l]-1));
col = RcppArmadillo::sample(colors, 1, 0)[0];
n1 = nei4(x, rows[k], cols[l], col);
if(log(runif(1)[0]) < (beta*(n1-n0))){
x(rows[k]-1, cols[l]-1) = col;
n0 = n1;
}
s = s + n0;

}
}
if(2*i > max_iter){
S = S + s;
}

}
return 2*S/max_iter;

}
```

6.1.5 Segmentación de imágenes

Después de todas estas secciones ya tenemos todos los materiales disponibles para la tarea que queríamos realizar. Consideraremos las imágenes como objetos estadísticos, pero no como tal, sino que consideraremos que tenemos una imagen con cierta distorsión y, es decir que el color de gris (o cualquier otra escala, por conveniencia) se presenta con cierta perturbación. El objetivo de la segmentación de imágenes es, dada esta imagen distorsionada, realizar un procedimiento de 'clustering' de píxeles atendiendo únicamente a la estructura de dependencia espacial de la estructura.

Esta dependencia espacial 'real' de píxeles se nota por x, donde generalmente y puede tomar valores reales positivos y x solo discretos, por conveniencia. Estamos pues interesados en conocer la distribución a posteriori de x dada y, es decir $\pi(x|y) \propto f(y|x)\pi(x)$. En esta expresión $f(y|x)$ representa la verosimilitud de los datos y la función link entre la imagen original y su clasificación, mientras que $\pi(x)$ representa nuestra información a priori (o propiedades deseadas) sobre el comportamiento de la imagen 'real'.

Generalmente esta distribución a priori suele ser el modelo de Potts ya estudiado con G categorías:

$$\pi(x|\beta) = \frac{1}{Z(\beta)} exp\left(\beta \sum_{i\sim j} I_{x_j=x_i}\right) \tag{6.11}$$

Dado x, y por facilidad computacional, asumimos que las observaciones en y son variables independientes normales. Esto se hace porque es más fácil parametrizar que una distribución multinomial que tome valores en $0, ..., 255$:

$$f(y|x,\sigma^2,\mu_1,...,\mu_G) = \prod_{i\in I} \frac{1}{(2\pi\sigma^2)^(1/2)} exp\left(-\frac{1}{2\sigma^2}(y_i-\mu_{x_i})\right) \tag{6.12}$$

Para los parámetros de estas normales se suelen utilizar distribuciones a priori uniformes:

$$\beta \sim U(0,2) \tag{6.13}$$
$$\mu \sim U(0 \leq \mu_1 \leq ... \leq \mu_G \leq 255) \tag{6.14}$$
$$\pi(\sigma^2) \propto \sigma^{-2} I_{[0,\infty)}(\sigma^2) \tag{6.15}$$

Esta última no es más que una distribución uniforme en el logaritmo de σ. Los μ_i no tienen por qué ordenarse, pero así evitan el problema del reordenamiento pivotal explicado en el capítulo de las mixturas. La distribución conjunta a posteriori es la siguiente:

$$\pi(x,\beta,\sigma^2,\mu|y) \propto \pi(\beta,\sigma^2,\mu) \times \frac{1}{Z(\beta)} exp\left(\beta \sum_{j\sim i} I_{x_j=x_i}\right) \times \prod_{i\in I} \frac{1}{(2\pi\sigma^2)^{1/2}} exp\left(\frac{-1}{2\sigma^2}(y_i-\mu_{x_i})^2\right) \tag{6.16}$$

Comenzamos a continuación a construír las diversas distribuciones totalmente condicionadas para el trabajo del muestreo de Gibbs. La de x_i:

$$P(x_i = g|y,\beta,\sigma^2,\mu) \propto exp\left(\beta \sum_{i\sim j} I_{x_j=g} - \frac{1}{2\sigma^2}(y_i-\mu_g)^2\right) \tag{6.17}$$

Aquí puede comprobarse que, una vez que x es conocido, los elementos de distinta categoría se separan y el resto de parámetros pueden simularse independientemente condicionados a x, y y σ^2. Si notamos por $n_g = \sum_{i\in I} I_{x_i=g}$ y $s_g = \sum_{i\in I} I_{x_i=g} y_i$, la distribución totalmente condicionada de μ_g es una distribución normal truncada en $[\mu_{g-1}, \mu_{g+1}]$ (por razones obvias, $\mu_0 = 0, \mu_{g+1} = 255$), de media s_g/n_g y varianza σ^2/n_g. La distribución condicionada de σ^2 es una gamma inversa con parámetros $|I|^2/2$ y $\sum_{i\in I}(y_i-\mu_{x_i})^2/2$.

Finalmente, la distribución totalmente condicionada de β es aquella que:

$$\pi(\beta|y) \propto \frac{1}{Z(\beta)} exp\left(\beta \sum_{j\sim i} I_{x_j=x_i}\right) \tag{6.18}$$

.

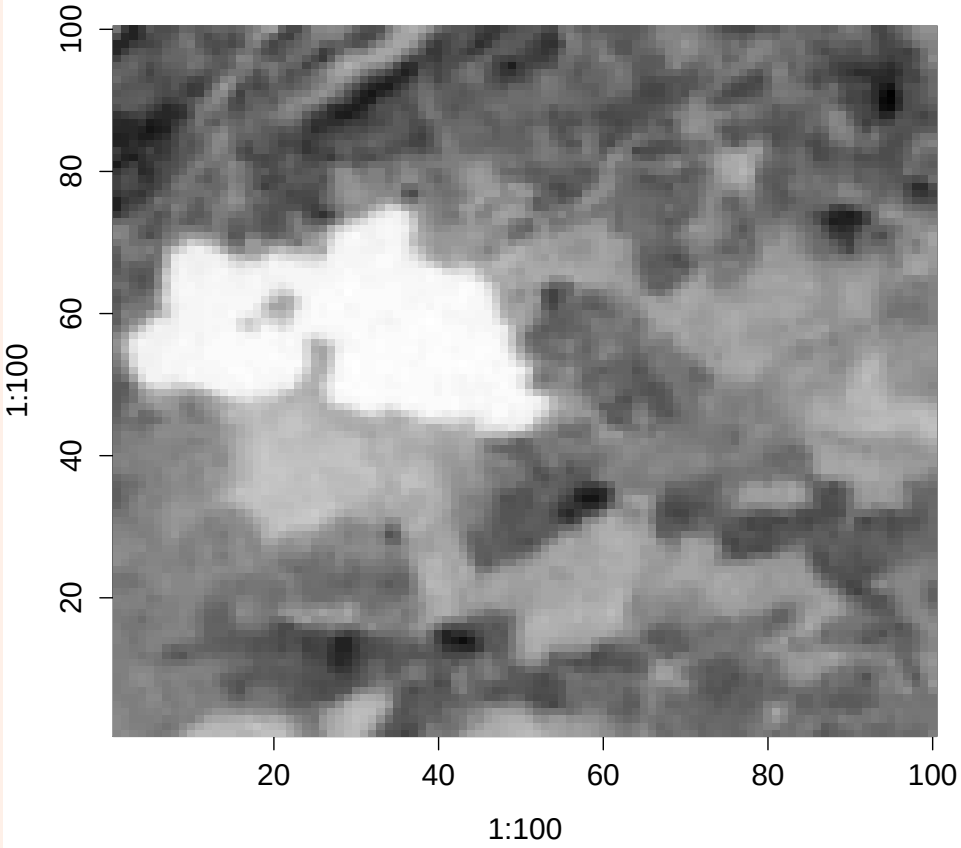
Ejercicio 6.4 Una implementación rápida en R de la segmentación de imágenes bayesiana podría ser la siguiente:

```
bay <- function (y, ncol=2, max_iter=10^3)
{
    numb = dim(y)[1]
    x = 0 * y
    mu = matrix(0, max_iter, 6)
    sigma2 = rep(0, max_iter)
    mu[1, ] = c(35, 50, 65, 84, 92, 120)
    sigma2[1] = 20
    beta = rep(1, max_iter)
```

```
    xcum = matrix(0, numb^2, 6)
    n = rep(0, 6)
    cat("Computing normalizing constant \n")
    Z = generatefbeta(y, max_iter, ncol)
    thefunc = approxfun(seq(0.1, 2, by = 0.1), Z)
    cat("Gibbs sampler on course \n")

    ## Main loop

    for (i in 2:max_iter) {
        cat("Iteration i", i, "\n")
        lvr = 0
        for (k in 1:numb) {
            for (l in 1:numb) {

                for(j in 1:ncol){
                    n[j] = nei4(x, k, l, j)
                }
                prob = exp(beta[i-1] * n) * dnorm(y[k, l],
                mu[i-1,], sqrt(sigma2[i-1]))

                if(NaN %in% prob){
                    cat("Hay un NA", "\n")
                    prob = exp(beta[i-1] * n)
                }

                x[k, l] = sample(1:ncol, 1, prob = prob)

                xcum[(k - 1) * numb + l, x[k, l]] = xcum[(k-1) *
numb + l, x[k, l]] + 1
                lvr = lvr + n[x[k, l]]
            }
        }

        ## Parameter simulation
        mu[i, 1] = truncnorm(1, mean(y[x == 1]),
         sqrt(sigma2[i - 1]/sum(x == 1)), 0, mu[i - 1, 2])

        for(j in 2:(ncol-1)){
            mu[i, j] = truncnorm(1, mean(y[x == j]),
         sqrt(sigma2[i - 1]/sum(x == j)), mu[i, j - 1], mu[i - 1, j + 1])
        }

        mu[i, ncol] = truncnorm(1, mean(y[x == ncol]),
 sqrt(sigma2[i-1]/sum(x == ncol)), mu[i, ncol-1], 255)

        sese = (y - mu[i, 1])^2 * (x == 1)
```

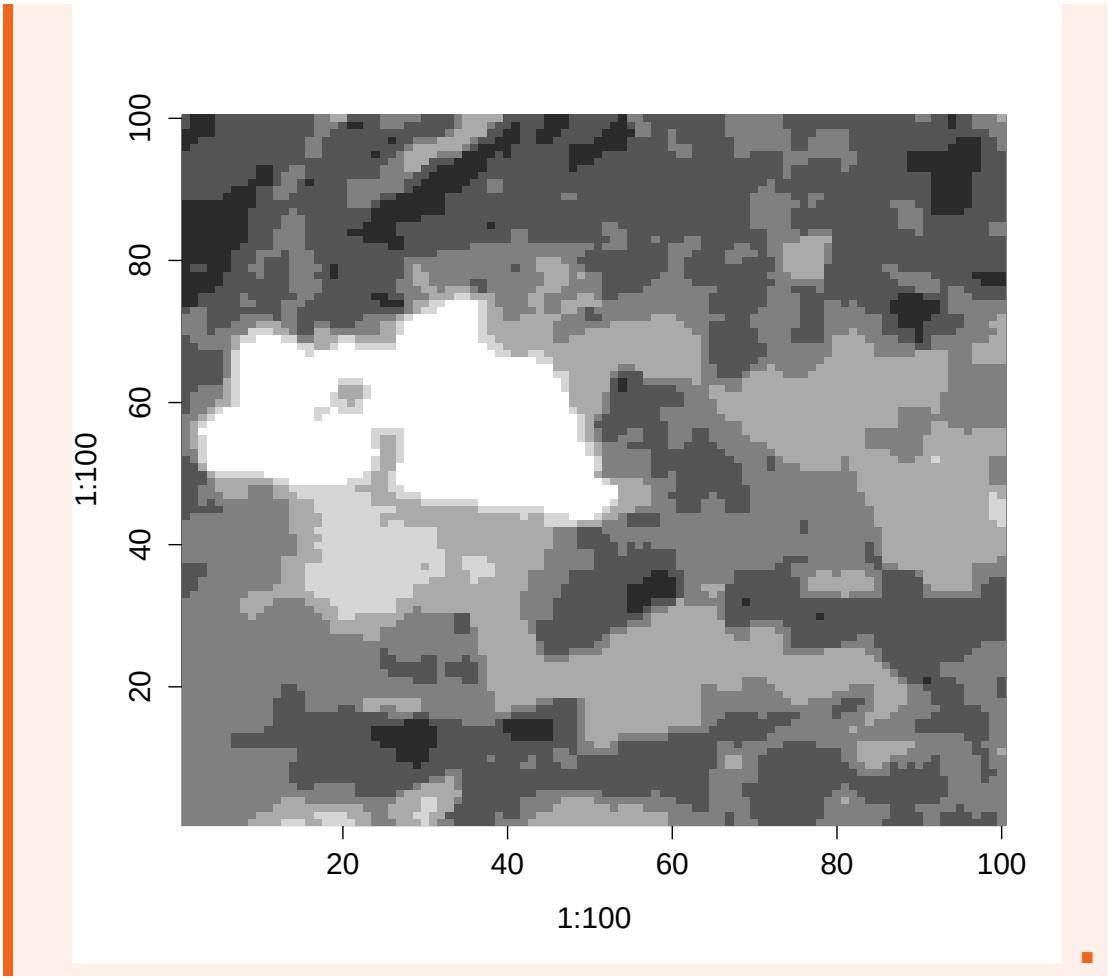
```
        for(j in 2:ncol){
            sese = sese + (y - mu[i, j])^2 * (x == j)
        }

        sese = sum(sese)

        if(is.nan(1/rgamma(1, (numb)^2/2, sese/2))){
            sigma2[i] = sigma2[i-1]
        }
        else{
             sigma2[i] = 1/rgamma(1, (numb)^2/2, sese/2)
        }



        betatilde = beta[i - 1] + runif(1, -0.05, 0.05)
        if(betatilde > 1.96){
            betatilde = betatilde - 0.1
        }
        laccept = lvr * (betatilde - beta[i - 1]) + integrate(thefunc,
            betatilde, beta[i - 1])$value
        if (runif(1) <= exp(laccept)){
            beta[i] = betatilde}
        else {beta[i] = beta[i - 1]}
    }
    list(beta = beta, mu = mu, sigma2 = sigma2, xcum = xcum)
}
```

El algoritmo es bastante general, pero puede producir problemas dependiendo de la imagen que queramos segmentar, debido a que se pueden producir NA's en la simulación, especialmente en la parte de la simulación de σ^2. No obstante, suponemos que tenemos una imagen de satélite de un lago, pero que es demasiado borrosa como para distinguir las distintas partes que la contienen. La imagen original es la siguiente:




```
require(bayess)
data(Menteith)
Menteith = as.matrix(Menteith)

system.time({mod = bay(Menteith, ncol=6, max_iter = 4)})
aff=apply(mod$xcum,1,affect)
aff=t(matrix(aff,100,100))
image(1:100,1:100,aff,col=gray(6:1/6),xlab="",ylab="")
```

Y siendo la imagen segmentada esta:


1:100
20
40
60
80
100
1:100
20
40
60
80
100

7. Imputación múltiple

Los datos ausentes plagan la investigación de muchos estudios estadísticos. Existen multitud de métodos de imputación, tanto simple (como sustituir con medidas de centralización), como múltiples, basados por ejemplo en el algoritmo EM o en ecuaciones estructurales. Estos métodos de imputación múltiple generalmente utilizan la información de otros valores de otras variables no ausentes para predecir aquellos que sí lo son de la variable de interés y.

En este capítulo estudiaremos un método de imputación múltiple llamado FCS (Especificación Completamente Condicionada o Ecuaciones Encadenadas), basado en simulaciones MCMC. En este capítulo el código en `R` a desarrollar será bastante más sencillo que en capítulos anteriores, ya que contamos con una implementación muy eficiente de esta técnica en el paquete `MICE`. (Multiple Imputation using Chained Equations). En este sentido, desarrollaremos la teoría y utilizaremos dicho paquete para resolver algunos ejemplos en los que la imputación múltiple puede tener sentido.

7.1 Imputación usando ecuaciones encadenadas

Para ponernos en situación notemos por y una matriz $n \times k$ de datos con n individuos y k variables. Notemos y_j a la j-ésima variable del estudio. A continuación nos conviene distinguir entre dos subconjuntos de nuestras variables, dependiendo de si un valor es observado o no. Denotamos por y_j^{obs} como la parte observada de y_j y y_j^{aus} como su parte ausente. Lo natural es que la imputación sobre y_j^{aus} venga dada por algún tipo de relación entre y_j y y_{-j} (todas las variables exceptuando la j-ésima). La naturaleza de esta relación tendría que averiguarse a través de la parte no ausente, es decir, a través de y_j^{obs}. Generalmente, este marco de actuación genera una serie de problemas:

- Los predictores y_{-j} podrían contener a su vez valores ausentes.
- Podría darse dependencia circular en el sentido que y_j^{aus} puede depender de y_h^{aus} y viceversa. $h \neq j$.
- Dependiendo del estudio, las variables pueden venir definidas por un orden (una serie temporal pej).
- Más importantemente, dentro de un marco de datos pueden encontrarse variables en muchas escalas diferentes, lo que dificulta un marco de actuación común.
- Las relaciones entre y_j y sus predictores puede ser de naturaleza compleja (no lineal).

Estos son solo algunos de los problemas que podrían surgir en nuestro análisis. Anteriormente, existía también un método de imputación múltiple basado en MCMC que los datos venían definidos por algún modelo de densidad multivariante $P(Y|\theta)$. La dificultad de este método es evidente en el momento que tenemos un marco de datos con variables de distinto tipo. Aunar todas esas variables bajo un mismo modelo teórico multivariante (como por ejemplo hace el algoritmo EM y la normal multivariante) es por un lado irreal y por otro difícil.

FCS es un método más natural en el sentido de que no partimos de una densidad multivariante. En su lugar la definimos implícitamente especificando una densidad univariante para cada variable del estilo $P(y_j|y_{-j},\theta_j)$. Esta densidad es la usamos para imputar y_j^{aus} dado y_{-j} mediante algún tipo de regresión (generalmente lineal o logística) sobre los casos y_j^{obs}. Se realizan tantos paseos sobre todas las densidades como iteraciones del algoritmo sean necesarias.

Las ventajas de FCS sobre otros métodos son evidentes: evitar tener que especificar directamente una distribución multivariante permite mucha flexibilidad a la hora de trabajar los datos. Es decir, convertimos un problema k dimensional en k problemas unidimensionales. Por otra parte, la idea de especificar un modelo de imputación diferente para cada variable es bastante natural. No obstante, FCS también cuenta con algunas dificultades. El tener que especificar un modelo diferente y adecuado para cada variable puede ser por un lado pesado para el investigador, y por otro es computacionalmente mucho más exigente. Por otra parte, evaluar la calidad de las imputaciones puede resultar difícil ya que la densidad conjunta teórica no tiene por qué existir.

En cualquier caso, comencemos a definir el método. Supongamos que tenemos $Y=(Y_1,...,Y_k)$ un vector de k variables aleatorias con una distribución $P(Y|\theta)$. Asumimos que la anterior distribución queda totalmente definida por θ. El procedimiento general comprendería los siguientes pasos:

1. Determinar la distribución a posteriori $p(\theta|y^{obs})$ de θ usando los datos observados y^{obs}.
2. Muestrear un valor θ^* de $p(\theta|y^{obs})$
3. Por último muestrear un valor y^* de $p(y^{aus}|y^{obs},\theta=\theta^*)$, la distribución condicional a posteriori de y^{aus} dado θ^*.

Generalmente este procedimiento es pesado y solamente se repiten los pasos 2 y 3 un número modesto de veces (generalmente suelen ser suficientes entre 5 y 10 iteraciones). Lógicamente desde el marco teórico univariante esto parece fácil, pero si Y es multivariante (caso $k>1$) lo lógico es utilizar un marco de simulación basado en el muestreo de Gibbs. El paso realmente complicado es el primero, donde necesitamos determinar la distribución a posteriori $p(\theta|y^{obs})$. En nuestro caso, aplicamos el muestreo de Gibbs muestreando de distribuciones condicionales de la forma:

$$P(Y_1|Y_{-1},\theta_1) \tag{7.1}$$
$$\dots \tag{7.2}$$
$$P(Y_k|Y_{-k},\theta_k) \tag{7.3}$$

Es decir, utilizamos los parámetros θ_i solo en las densidades univariantes donde intervienen respectivamente. En otras palabras, estos parámetros no serían necesariamente el producto de una distribución conjunta $P(Y|\theta)$. De manera más clara, una iteración del muestreo de Gibbs comprendería los siguientes pasos:

$$\theta_1^* \sim P(\theta_1 | y_1^{obs}, y_2^{t-1}, ..., y_k^{t-1}) \tag{7.4}$$
$$y_1^{*(t)} \sim P(y_1^{aus} | y_1^{obs}, y_2^{t-1}, ..., y_k^{t-1}, \theta_1^{*(t)}) \tag{7.5}$$
$$\dots \tag{7.6}$$
$$\theta_k^* \sim P(\theta_k | y_k^{obs}, y_2^{t}, ..., y_{k-1}^{t}) \tag{7.7}$$
$$y_k^{*(t)} \sim P(y_k^{aus} | y_k^{obs}, y_2^{t}, ..., y_{k-1}^{t}, \theta_k^{*(t)}) \tag{7.8}$$
$$\tag{7.9}$$

Y si nos fijamos, en realidad ya podemos recurrir a modelos de imputación donde y es univariante para muestrear de las anteriores distribuciones. En la siguiente sección se detallan varios modelos dependiendo de la escala de la variable, con lo que el problema quedaría completamente solucionado.

En ningún caso se utiliza información (cuál se iba a usar, en cualquier caso) de y_j^{aus} para muestrear de $\theta_j^{*(t)}$. Las simulaciones del modelo pueden ser paralelizadas fácilmente.

7.1.1 Modelos de imputación univariante

Para los pasos del muestreo de Gibbs es necesario recurrir a modelos de imputación univariante. Dependiendo de la distribución de $y|x$, los algoritmos son los siguientes:

Datos normalmente distribuídos (escala)

Para este caso, recurrimos a un modelo de regresión lineal ordinario. Suponemos media βx, varianza σ^2 como parámetros. Por otra parte descomponemos $x = (x^{obs}, x^{mis})$ como la matriz $n \times k$ de datos y $n = n^{obs} + n^{aus}$ para la variable a predecir. El algoritmo consta de los siguientes pasos:

1. Estimar β por una regresión ordinaria lineal de y sobre x. Concretamente por: $b = (x^{obs'} x^{obs})^{-1} x^{obs'} y^{obs}$
2. Muestrear $g \sim \chi^2(n^{obs} - p)$
3. Estimar $\sigma^{2*} = (y^{obs} - x^{obs} b)'(y^{obs} - x^{obs} b/g)$
4. Muestrear $w_1 \sim N(0, I_k)$
5. Calcular $b* = b + \sigma^{2*} w_1 V^{1/2}$, donde $V^{1/2}$ es la matriz triangular superior de una descomposición Cholesky de $V = (x^{obs'} x^{obs})^{-1}$
6. Muestrear $w_2 \sim N(0, I_n^{aus})$, donde este último término representa una matriz triangular de tamaño n donde los elementos donde y_i esté ausente valen 1 y 0 en caso contrario.
7. Imputar $y^* = x^{aus} b^* + w_2 \sigma^*$

Este algoritmo debería ser bastante robusto a falta de normalidad, pero no obstante se proponen un par de alternativas para tener en cuenta este fenómeno:

Predictive mean matching. Basta sustituir el último paso calculando $y^{aus} = x^{aus} b^*$. Para cada valor ausente, imputar ese valor por el valor de $y^{obs} = x^{obs} b^*$ que más cercano se encuentre a y^{aus}.

Hot-deck Reemplazar el penúltimo paso el muestreo por uno de n^{aus} valores con reemplazamiento del conjunto de n^{obs} residuos estandarizados.

Datos de respuesta binaria

Recurrimos a un modelo de regresión logística. Asumimos $P(y_i|\beta, x_i) = \frac{e^{x_i\beta}}{(1+e^{x_i\beta})^{y_i}}\left(1 - \frac{e^{x_i\beta}}{(1+e^{x_i\beta})^{1-y_i}}\right)$.

A continuación, las imputaciones se obtienen de la siguiente manera:

1. Obtener por un método numérico habitual (Newton-Raphson por ejemplo) un estimador b de β y de $Var[\beta]$ (un hessiano) usando solo los datos completos.
2. Muestrear $b^* \sim N(b, Var[b])$
3. Para cada observación ausente, calcular $w_i = \frac{e^{x_i b^*}}{1+e^{x_i b^*}}$
4. Para cada observación ausente, muestreamos un $u_i \sim U(0,1)$. Si $u_i > w_i$, imputamos esa observación como $y_i = 1$, en otro caso, $y_i = 0$.

Datos categóricos

Este caso es el más sencillo porque no es más que una generalización natural del anterior. Notemos las categorías por $0, ..., s-1$. La distribución de y puede caracterizarse por $\log(P(y = j|x)/P(y = 0|x)) = \beta_j x$. En este sentido, el modelo para y no es más que una serie de regresiones logísticas apiladas considerando una categoría base (one vs rest).

1. De manera similar, utilizar un modelo de regresión logística multinomial, estimar $b = \hat{\beta}$ y $Var(b) = \hat{Var}(\beta)$.
2. Muestrear $b^* \sim N(b, Var(b))$
3. Para cada observación ausente, calcular $\pi_{i,j}^{aus} = \frac{e^{-b_j^* x_i}}{1+\sum_{v=1}^{s-1} e^{b_v^* x_i}}$
4. Para cada observación ausente x_i, muestrear de $0, ..., s-1$ con probabilidad $\pi_{i,j}^{aus}$.

7.2 Probando el algoritmo

En esta corta sección nos dedicaremos a probar con varios marcos de datos de distinta naturaleza cómo se comporta el algoritmo a nivel de calidad de imputaciones.

Ejercicio 7.1 En primer lugar probaremos con el marco de datos `mtcars` del paquete `datasets`. Este marco se caracteriza por tener variables de todo tipo, por lo que se presta muy bien a analizar los errores cometidos. Por otra parte, es un marco de datos pequeños, por lo que servirá para evaluar la calidad de las imputaciones cuando hay pocas observaciones involucradas.

```
library(mice)

data = mtcars
data$am = factor(data$am)
data$vs = factor(data$vs)

datosoriginales = data
```

Introducimos de manera aleatoria datos ausentes:

```
n = nrow(data)
m = ncol(data)

p = 0.15
```

```
## Esperamos aprox n*m*p datos ausentes

for(i in 1:n){
  for(j in 1:m){
    u = runif(1)
    if(u < p){
      data[i,j] = NA
    }
  }
}

expected = n*m*p
length(which(is.na(data)))
```

Corremos el algoritmo, usando para todas las variables Predictive Mean Matching:

```
imputation = mice(data, m=10, maxit=100, method = rep("fastpmm", 11))
completos = complete(imputation, action = 5)
```

Finalmente, evaluamos la calidad de las imputaciones mediante la raíz del error cuadrático medio (para las observaciones categóricas no obstante carece de sentido esta medida).

```
a = as.numeric(completos[is.na(data)])
b = as.numeric(datosoriginales[is.na(data)])

rmse = sqrt(mean((a-b)^2))
```

Otro ejemplo también con variables de distinto tipo muy sencillo es el siguiente:

```
data(nhanes2)
head(nhanes2)

imputation = mice(nhanes2, m=10, maxit=100, method = rep("fastpmm", 4))
completos = complete(imputation, action = 5)

print(completos)
```

■

8. Minería de texto

8.1 Descubrimiento de temáticas

En este capítulo abordaremos un problema relativamente reciente de la Inteligencia Artificial. Supongamos que tenemos una gran cantidad de documentos que analizar, con el mismo o diferentes propósitos. Una de las problemáticas que nos puede surgir entre tanta información es la de buscar temáticas comunes entre diversos documentos. Esto puede tener mucha utilidad en diversos campos, pero especialmente en motores de búsqueda, donde las páginas (especialmente de noticias), quedan indexadas según una serie de palabras clave. Éstas serían posteriormente agrupadas automáticamente por el mismo motor para proporcionar información de una manera ordenada.

En definitiva, nuestro problema explicado de manera informal es el siguiente: dados una serie de documentos de texto cualesquiera, determinar si existen k temáticas relativas a esos documentos. Este número k de temáticas viene fijado de antemano. En los últimos años se han desarrollado multitud de técnicas para intentar solventar este problema. Para este trabajo vamos a centrarnos en una llamada Latent Dirichlet Allocation (que traduciremos por Asignación latente de Dirichlet), cuya estimación de parámetros puede ser solucionada por técnicas MCMC. Nótese que estas técnicas no son las únicas para resolver el problema, pero como veremos se prestan a ser las más fáciles de implementar en este caso.

La asignación latente de Dirichlet (la llamaremos LDA por comodidad a partir de ahora) está implementada en varios lenguajes de programación, siendo la más fácil de utilizar una de Python. Ésta, además implementa dicha técnica usando el muestreo de Gibbs (concretamente una versión llamada muestreo de Gibbs colapsado), por lo que resulta especialmente adecuada para ejemplificar.

8.2 Asignación latente de Dirichlet

Antes de comenzar a explicar de manera formal en qué consiste dicha técnica, vamos a ver cómo funciona de manera constructiva: LDA supone que los documentos son mixturas de temáticas, y que las temáticas a su vez son mixturas de palabras. Es decir, supongamos en primer lugar que queremos generar un documento con respecto a estas suposiciones:

1. Decidiríamos en primer lugar cuántas palabras N va a tener un documento, por ejemplo de acuerdo con una distribución de Poisson.
2. Decidimos una distribución para las temáticas, por ejemplo de acuerdo con una distribución Dirichlet.
3. Ahora generamos cada palabra del documento de la manera siguiente:
   a) Decidimos la temática de la palabra, escogiendo una de acuerdo a una distribución multinomial, conjugada con la Dirichlet anterior.
   b) Una vez escogida la temática, escogemos la palabra de acuerdo a la distribución multinomial de la temática.

8.2.1 Formalización de la generación

Definamos lo siguiente:

- Una palabra es la unidad básica de dato discreto, definida como un miembro de vocabulario indexado entre $1,...,V$. Representamos las palabras como vectores unitarios, que tienen un solo componente igual a 1 y el resto igual a 0. La v-ésima palabra del vocabulario queda representada por $w^v = 1$ solo para v.
- Un documento es una secuencia de N palabras denotadas por $w = (w_1,...,w_N)$.
- Un cuerpo es una colección de M documentos denotados por $D = w_1,...,w_M$

De manera formal, para generar un documento lo que haríamos sería lo siguiente:

1. Escoger $N \sim Poisson(\varepsilon)$
2. Escogemos $\theta \sim Dir(\alpha)$
3. Para cada una de las N palabras w_i:
   a) Escogemos una temática según $z_n \sim Multinomial(\theta)$
   b) Escogemos una palabra w_i con probabilidad $P(w_i|z_i,\beta)$.

Realizaremos algunas hipótesis de simplificación para continuar. Para empezar y como ya hemos comentado, la dimensionalidad k de la distribución Dirichlet queda fijada de antemano (y por tanto el número de temáticas a encontrar). En segundo lugar, las probabilidades de cada una de las palabras quedan parametrizadas por una matriz β de dimensiones $k \times V$, donde $\beta_{ij} = P(w^j = 1|z^j = 1)$. En un principio trataremos esta matriz como algo fijado que tendremos que estimar eventualmente. Por último, la suposición de que los documentos tienen un número de palabras de acuerdo con una distribución de Poisson puede ser ignorada completamente.

8.3 Inferencia y estimación usando muestreo de Gibbs colapsado

Hasta ahora hemos aprendido cómo generar documentos de acuerdo con este modelo. Hemos empezado así porque LDA es un modelo generativo, en el sentido de que los documentos vienen generados según esa serie de hipótesis para luego realizar el aprendizaje de manera más sencilla. En primer lugar explicaremos de manera similar a como hemos hecho antes el procedimiento de aprendizaje de manera informal, y luego lo formalizaremos de manera más detallada.

Nuestro caso ahora se centra en que tenemos una serie de documentos a los cuales queremos extraer temáticas:

- De manera aleatoria, para cada cada palabra en cada documento, asignamos una temática.
- Esta asignación ya nos da una representación de la distribución de las temáticas y de las palabras (si bien nada buenas). Para mejorarlas:
  - Para cada palabra en cada documento y para cada temática calcular 1) $P(\text{temtica } t|\text{documento } d) =$ la proporción de palabras en el documento d que están asignadas a la temática t y 2) $P(\text{palabra } w|\text{temtica } t) =$ la proporción de asignaciones a la temática t debidas a esta palabra w.

- Reasignar a la palabra una nueva temática de acuerdo con $P(\text{temtica } t|\text{documento } d) \times P(\text{palabra } w|\text{temtica } t)$ para todas las temáticas t

Después de repetir este procedimiento un número elevado de veces, alcanzaremos un estado estable en el que las asignaciones apenas cambian. En este estado ya podemos usar las asignaciones para estimar las mixturas en cada documento (simplemente contando las palabras asignadas a cada temática en cada documento) y las palabras asociadas a cada temática (contando el total de palabras asignadas a cada temática).

8.3.1 Formalización del aprendizaje

Dados D documentos conteniendo T temáticas expresadas bajo W palabras únicas, podemos representar $P(w|z)$ como un conjunto de T distribuciones multinomiales ϕ sobre las W palabras, de tal manera que $P(w|z=j) = \phi_w^{(j)}$ y equivalentemente, representamos $P(z)$ como D distribuciones multinomiales sobre las T temáticas, de tal manera que para un documento cualquiera $P(z=j) = \theta_j^{(d)}$.

Nuestra estrategia para descubrir las temáticas pasa por explorar la distribución a posteriori de las temáticas sobre las palabras $P(z|w)$. Una vez evaluada, podremos obtener estimaciones tanto de ϕ como de θ. (Estas dos últimas pueden ser representadas como matrices por conveniencia). Utilizamos el muestreo de Gibbs para muestrear de esta distribución a posteriori, para ello, definimos lo siguiente:

$$\begin{gathered} w_i|z_i, \phi^{(z_i)} \sim Multinomial(\phi^{(z_i)}) \\ \phi \sim Dirichlet(\beta) \\ z_i|\theta^{(d_i)} \sim Multinomial(\theta^{(d_i)}) \\ \theta \sim Dirichlet(\alpha) \end{gathered}$$

Donde α y β son hiperparámetros para controlar las distribuciones a priori de ϕ y θ. Estas distribuciones a priori son conjugadas con la Multinomial, permitiendo calcular la distribución conjunta $P(w,z) = P(w|z)P(z)$. Si en esta distribución integramos primero sobre ϕ obtendremos:

$$P(w|z) = \left(\frac{\Gamma(W\beta)}{\Gamma(\beta)^W}\right)^T \prod_{j=1}^{T} \frac{\prod_w \Gamma(n_j^{(w)} + \beta)}{\Gamma(n_j^{(\cdot)} + W\beta)} \tag{8.1}$$

donde $n_j^{(w)}$ es el número de veces que una palabra w ha sido asignada a la temática j en el vector de asignaciones z. Si en el segundo térmnino integramos sobre θ, obtendríamos:

$$P(z) = \left(\frac{\Gamma(T\alpha)}{\Gamma(\alpha)^T}\right)^D \prod_{d=1}^{D} \frac{\prod_j \Gamma(n_j^{(d)} + \alpha)}{\Gamma(n_\cdot^{(d)} + T\alpha)} \tag{8.2}$$

donde $n_j^{(d)}$ es el número de veces que una determinada palabra de un documento d ha sido asignada al documento j. Por tanto, nuestro objetivo sería evaluar la distribución a posteriori:

$$P(z|w) = \frac{P(w,z)}{\sum_z P(w,z)} \tag{8.3}$$

Desgraciadamente, esta distribución no puede ser calculada directamente, ya que su cálculo implica la evaluación de una distribución de probabilidad sobre un espacio de estados discretos muy amplios. No obstante, podemos usar el muestreo de Gibss, para obtener la expresión,

mediante cancelación de términos de las dos distribuciones anteriores, obteniendo la siguiente distribución totalmente condicionada:

$$P(z_i = j|z_{-i}, w) \propto \frac{n_{-i,j}^{(w_i)} + \beta}{n_{-i,j}^{(\cdot)} + W\beta} \frac{n_{-i,j}^{(d_i)} + \alpha}{n_{-i}^{(di)} + T\alpha} \tag{8.4}$$

donde $n_{-i}^{\cdot}$ es un conteo sin contar la asignación actual de z_i. Este resultado es bastante intuitivo, ya que de la expresión anterior el primer término expresa la probabilidad de w_i bajo la temática j, y el segundo ratio expresa la probabilidad de la temática $j$sobre el documento d_i. Una vez obtenida esta distribución totalmente condicionada, el algoritmo procede como sigue: Inicializamos todas las palabras de los documentos (realmente sólo aquellas incluídas entre las W del vocabulario) con valores entre 1 y T. A continuación asignamos temáticas a las palabras de acuerdo a las probabilidades obtenidas con la expresión anterior. Lo más adecuado, es que en cada iteración dichas probabilidades sean halladas únicamente con aquellas palabras cuya asignación haya sido realizado anteriormente. Una vez que la cadena ha corrido un número suficientemente alto de veces, llegará a un estado suficientemente estable (o estacionario).

Con un conjunto de muestras de la distribución a posteriori ya tendríamos resuelto el problema de asignación. No obstante, si estuviéramos interesados en conocer la forma de los parámetros distribucionales podríamos estimarlos mediante dicha muestra de la siguiente manera:

$$\hat{\phi_j^{(w)}} = \frac{n_j^{(w)} + \beta}{n_j^{(\cdot)} + W\beta} \tag{8.5}$$

$$\hat{\theta_j^{(d)}} = \frac{n_j^{(d)} + \alpha}{n_{\cdot}^{(d)} + T\alpha} \tag{8.6}$$

Escogiendo los parámetros α, β y el número de temáticas T

El algoritmo descrito en esta sección podría extenderse al caso en el que α y β fueran a su vez distribuciones dependientes de más hiperparámetros y muestrear de ellos, pero realizarlo realmente contribuye a un mayor coste computacional sin obtener mejoras de rendimiento. La elección de estos dos parámetros es importante a la hora de obtener distintos resultados: en particular, un β más grande produce un menos número de temáticas más amplias. Dados valores fijos a estos hiperparámetros se trataría de escoger un número de temáticas apropiado, lo que realmente es un problema de selección de modelos bayesiano. Lo natural en estadística bayesiana es evaluar la distribución a posteriori de los modelos dados los datos, y escoger aquel que de mayor densidad de probabilidad. En nuestro caso, nuestros datos son las palabras en los documentos, luego necesitaríamos evaluar la verosimilitud $P(w|T)$.

De nuevo, este es un problema intratable desde el punto de vista computacional, pero podríamos aproximar dicha distribución tomando la media harmónica de un conjunto de valores de $P(w|z,T)$ cuando z se muestrea de la distribución a posteriori $P(z|w,T)$. Estas últimas pueden hallarse usando 8.1. Una vez aproximada, podemos utilizar esta cantidad para escoger el número de temáticas adecuadamente.

8.4 Ejemplos de aplicación

En esta sección nos dedicaremos a aplicar este algoritmo en algunas situaciones interesantes en las que tenga sentido. Desde el punto de vista práctico estaremos utilizando el paquete `lda` disponible en `PyPI`, tanto bajo `Python2` como `Python3`. En este caso utilizaremos esta última implementación por evitar problemas con la autenticación via `oauth2`.

8.4.1 Análisis de perfil en Twitter

Con el auge de las nuevas redes sociales, un nuevo mundo de información textual se abre. Esta información es fácilmente accesible para cualquiera con una cuenta de por ejemplo Twitter (con algunas limitaciones) desde cualquier lenguaje de programación moderno. En nuestro caso, lo que vamos a realizar con el siguiente código es un análisis de perfil de una determinada cuenta de esta red social. Concretamente, estaríamos interesados en conocer el contenido en cuanto a temáticas de los tweets de una determinada persona.

Centrémonos en analizar de qué habla Elon Musk, CEO de Tesla Motors. Para ello necesitamos en primer lugar una serie de claves de acceso a la API de Twitter, las cuales son obtenibles fácilmente a través de la página web. Comencemos el análisis.

Ejercicio 8.1

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
Latent Dirichlet Allocation
Twitter Crawler

@author: JJimenez
"""
import numpy as np

access_token = "xxxxx-xxxxxx"
access_token_secret = "xxxxx"
consumer_key = "xxxxxxxxxxxxxxxxxxxxxxxxx"
consumer_secret = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"

import tweepy

auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_token, access_token_secret)

api = tweepy.API(auth)

public_tweets = api.user_timeline('elonmusk', count = 1000)

texts = []
for tweet in public_tweets:
    texts.append(tweet.text)

from sklearn import feature_extraction


paravocab = feature_extraction.text.CountVectorizer(stop_words = 'english',
                                                    max_features = 50).fit(texts)

vocab = paravocab.vocabulary_
vocab = vocab.keys()
```

```
vocab = list(vocab)
vocab.sort()

matriz = feature_extraction.text.CountVectorizer(stop_words = 'english',
                                                 vocabulary = vocab).fit_transform(texts

import lda

model = lda.LDA(n_topics=5, n_iter=1500, random_state=1)
model.fit(matriz)

topic_word = model.topic_word_
n_top_words = 5

for i, topic_dist in enumerate(topic_word):
    topic_words = np.array(vocab)[np.argsort(topic_dist)][:-n_top_words:-1]
    print('Topic {}: {}'.format(i, ' '.join(topic_words)))
```

Ejecutar el anterior código produce el siguiente resultado (las siguientes son las palabras más comunes para cada temática):

```
Topic 0: good rocket time station
Topic 1: http like flight https
Topic 2: http tesla rt cars
Topic 3: landing rocket ship just
Topic 4: http rt spacex launch
```

Ninguna sorpresa. A Elon Musk le gusta hablar en su cuenta personal de Twitter de varias temáticas: los lanzamientos de los cohetes SpaceX y de los coches que fabrica su compañía. Por otra parte es notable comprobar que tiene por costumbre añadir en sus tweets bastantes enlaces, ya sean imágenes u otro contenido externo.

■

8.4.2 Temática según autores clásicos

Nos preguntamos si LDA sería capaz de captar una serie de temáticas relativas a varios clásicos de la literatura anglosajona. La elección del inglés para realizar este apartado es parecida a la del apartado anterior: contamos con un diccionario de palabras frecuentes en el idioma (como preposiciones, determinantes...) a excluír. Nótese que si incluyésemos este tipo de palabras en el análisis, probablemente las palabras más frecuentes en cada temática serían ellas, haciendo imposible distinguir entre las mismas.

Para el análisis utilizaremos dos libros de Jane Austen, Sentido y Sensibilidad y Orgullo y Prejuicio, dos de Charles Dickens y Moby Dick, de Herman Melville. Utilizaremos versiones en `.txt` de `Project Gutenberg`. que bajaremos secuencialmente con el código.

Ejercicio 8.2

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
Created on Sun May  3 01:43:34 2015
```

```
Book recurrent themes
@author: JJimenez
"""
import urllib2
import numpy as np
from sklearn import feature_extraction
import lda
from bs4 import BeautifulSoup
import re

titles = ['Pride and Prejudice', 'Sense and Sensibility', 'Oliver Twist',
          'Christmas Carol', 'Moby Dick']
books = ['https://www.gutenberg.org/files/42671/42671-h/42671-h.htm',
         'https://www.gutenberg.org/files/21839/21839-h/21839-h.htm',
         'https://www.gutenberg.org/files/730/730-h/730-h.htm',
         'https://www.gutenberg.org/files/46/46-h/46-h.htm',
         'https://www.gutenberg.org/files/2701/2701-h/2701-h.htm']

texts = []

for link in books:
    parse = urllib2.urlopen(link)
    soup = BeautifulSoup(parse)
    texto = soup.get_text()
    texto = texto.replace("\t", "").replace("\r", "").replace("\n", "")
    texto = re.sub('[^a-zA-Z]'," ", texto)
    texts.append(texto)

paravocab = feature_extraction.text.CountVectorizer(stop_words = 'english',
                                                    max_features = 1000).fit(texts)

vocab = paravocab.vocabulary_
vocab = vocab.keys()
vocab = list(vocab)
vocab.sort()

matriz = feature_extraction.text.CountVectorizer(stop_words = 'english',
                                                 vocabulary = vocab).fit_transform(texts

model = lda.LDA(n_topics=10, n_iter=1500, random_state=1)
model.fit(matriz)

topic_word = model.topic_word_
n_top_words = 10

for i, topic_dist in enumerate(topic_word):
    topic_words = np.array(vocab)[np.argsort(topic_dist)][:-n_top_words:-1]
    print('Topic {}: {}'.format(i, ' '.join(topic_words)))
```

Bibliografía

Bibliografía

[Bro98] Stephen P Brooks. “Markov chain Monte Carlo method and its application”. En: *The Statistician* 47.1 (1998), páginas 69-100. ISSN: 0039-0526. DOI: 10.2307/2988428.

[Cor+] Bayesian Core y col. “Mixture Models”. En: (), páginas 291-343. ISSN: 1367-4811.

[DD05] Frank Dellaert y Frank Dellaert. “Markov Chain Monte Carlo Basics”. En: *October* October (2005), páginas 1-14.

[Dir+14] Title Dirichlet y col. “Package ‘ dpmixsim ’”. En: (2014).

[FNT10] Colin Fox, Geoff K Nicholls y Sze M Tan. “An Introduction To Inverse Problems An Introduction To Inverse Problems”. En: *Statistics* (2010).

[Gam] Dani Gamerman. “Statistical Models for Space-time Data”. En: ().

[Gey12] Charles J Geyer. “MCMC Package Example (Version 0 . 9-1)”. En: 1 (2012), páginas 1-16.

[GRS96] W R Gilks, S Richardson y D J Spiegelhalter. *Markov Chain Monte Carlo in Practice*. 1996. DOI: 10.2307/1271145. URL: http://www.jstor.org/stable/1271145?origin=crossref.

[GL10] B Grün y F Leisch. “BayesMix: an R package for Bayesian mixture modeling”. En: *Technique Report* (2010), páginas 1-11. URL: http://www.ci.tuwien.ac.at/~gruen/BayesMix/bayesmix-intro.pdf.

[Had10] Jarrod D Hadfield. “MCMC methods for multi-respoinse generalized linear mixed models: The MCMCglmm R package”. En: *Journal of Statistical Software* 33.Brown (2010), páginas 1-22.

[HMR13] Ghassan Hamra, Richard MacLehose y David Richardson. “Markov chain monte carlo: An introduction for epidemiologists”. En: *International Journal of Epidemiology* 42 (2013), páginas 627-634. ISSN: 03005771. DOI: 10.1093/ije/dyt043.

[MMR05] Jean Michel Marin, Kerrie Mengersen y Christian P. Robert. “Bayesian Modelling and Inference on Mixtures of Distributions”. En: *Handbook of Statistics* 25 (2005), páginas 459-507. ISSN: 01697161. DOI: 10.1016/S0169-7161(05)25016-2. arXiv: 0804.2413.

[MR14] Jean-Michel Marin y Christian P Robert. *Bayesian Essentials with R*. 2014, página 296. ISBN: 978-1-4614-8686-2. DOI: 10.1007/978-1-4614-8687-9. URL: http://link.springer.com/10.1007/978-1-4614-8687-9.

[Mcm05] Data-driven Mcmc. "MCMC Tutorial at ICCV Motivation for MCMC". En: October (2005).

[MJG12] Morphometric Mcmc, Leif T Johnson y Charles J Geyer. "Morphometric MCMC (mcmc Package Ver. 0.9)". En: (2012), páginas 1-20.

[Mcm+05] Trans-dimensional Mcmc y col. "References • Peter Green , Reversible jump Markov chain Monte Carlo , in " Highly Structured Stochastic Systems ", 2003 • Paskin & Thrun , Robotic Mapping with Polygonal Outline Model Selection • Most common case : inference on p (x | z), x continuous p". En: *October* October (2005), páginas 1-12.

[Ml] An Ml. "Mixture models (Ch . 16) Mixture models (cont ' d) Bayesian estimation (cont ' d) Mixture models - missing data". En: 1 ().

[Ric+14] Author Richard y col. "Package ' BayesFactor '". En: (2014).

[Sah00] Sujit Sahu. "Markov Chain Monte Carlo (MCMC) Introduction". En: August (2000).

[VA13] Sunay Vaishnav y Primary Advisor. "A Markov Chain Monte Carlo based approach to Image Segmentation". En: (2013).

[WG00] Michael D Ward y Kristian Skrede Gleditsch. "Location , Location , Location : An MCMC Approach to Modeling Spatial Context with Categorical Variables in the Study and Prediction of War 1". En: (2000).

[Zhu+05] Song-chun Zhu y col. "Markov Chain Monte Carlo for Computer Vision (SLIDES)". En: *October* October (2005).

[ZS02] Zhuowen Tu y Song-Chun Zhu. "Image segmentation by data-driven markov chain monte carlo". En: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 24.5 (2002), páginas 657-673. ISSN: 01628828. DOI: 10.1109/34.1000239.